VISIT…

WILEY

WILEY
XML Essentials

XPath Essentials

TEAM-Y

Andrew Watt

XPath Essentials

Andrew Watt

Wiley Computer Publishing

John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

XPath Essentials

XPath Essentials

Andrew Watt

Wiley Computer Publishing

John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

Publisher: Robert Ipsen
Editor: Cary Sullivan
Developmental Editor: Scott Amerman
Associate Managing Editor: Penny Linskey
New Media Editor: Brian Snapp
Text Design & Composition: Publishers' Design and Production Services, Inc.

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This book is printed on acid-free paper. ♾

Copyright © 2002 by Andrew Watt. All rights reserved.

Published by John Wiley & Sons, Inc., New York

Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ @ WILEY.COM.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-20548-6

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Dedicated to the memory of my late father,
George Alec Watt, a very special human being.

Contents

Introduction xxi
Acknowledgments xxv

Chapter 1 Starting with XML and XSLT 1

Overview of XML 1

A Simple XML Document 2
Structure of an XML Document 4
Logical 4
Elements 4
Types 5
Attribute Specification 5
Physical 7
Document Entity 7
Character and Entity References 7
Entity Declarations 8
Parsed Entities 8
Parameter Entities 8
Predefined Entities 9
Parsed Character Data 9
Notations 9
Syntax 10
XML Declaration and the Prolog 10
Comments 10
Processing Instructions 11
Elements 11
CDATA Sections 11

Language Identification 12
Well-formed and Valid 12
Well-Formed 12
Valid and Non-Valid 12
Normalization 13
XML Uses 14
The XML Technologies Jigsaw 16
Putting XML to Work 18
XML Namespaces 19
Where Does XPath Fit? 22
What Can XPath Do? 22
What Are the Next Steps? 23
Knowledge Needed 23
Software 23
Where to Get Help 24
Introduction to XSLT 24
XPath in XSLT 28
The <xsl:stylesheet> Element 28
XSLT Top-Level Elements 29
The <xsl:import> Element 30
The <xsl:attribute-set> Element 30
The <xsl:decimal-format> Element 30
The <xsl:include> Element 31
The <xsl:key> element 31
The <xsl:namespace-alias> Element 31
The <xsl:output> Element 32
The <xsl:param> Element 33
The <xsl:preserve-space> Element 34
The <xsl:strip-space> Element 34
The <xsl:template> Element 34
The <xsl:variable> Element 35
Other XSLT Elements 36
The <xsl:apply-imports> Element 36
The <xsl:apply-templates> Element 36
The <xsl:attribute> Element 37
The <xsl:call-template> Element 38
The <xsl:choose> Element 39
The <xsl:comment> Element 40
The <xsl:copy> Element 40
The <xsl:copy-of> Element 41
The <xsl:element> Element 41
The <xsl:fallback> Element 42
The <xsl:for-each> Element 43
The <xsl:if> Element 43
The <xsl:number> Element 46

FREEMFLY

The <xsl:otherwise> Element 47
The <xsl:processing-instruction> Element 47
The <xsl:sort> Element 47
The <xsl:text> Element 48
The <xsl:value-of> Element 48
The <xsl:when> Element 48
The <xsl:with-param> Element 48
Attribute Value Templates 48
XSLT Tools 49
Saxon and Instant Saxon 49
Xalan 52
Microsoft MSXML3 53
Oracle XML Development Kit 54
Looking Ahead 54

Chapter 2 XPath Fundamentals 55

XPath Overview 56
The XPath Forms of Syntax 57
The XPath Data Model 60
XPath Nodes 63
XPath Expressions 64
Context Node 65
Current Node 71
Function Calls 72
XPath Location Paths 75
Location Steps 80
Axes 82
Child Axis 83
Descendant Axis 84
Parent Axis 87
Ancestor Axis 87
Following-Sibling Axis 88
Preceding-Sibling Axis 90
Following Axis 91
Preceding Axis 93
Attribute Axis 95
Namespace Axis 96
Self Axis 98
Descendant-Or-Self Axis 98
Ancestor-Or-Self Axis 98
Node Tests 98
Predicates 100
XPath Functions 104
Node Set Functions 104

The count ( ) Function 104
The id ( ) Function 105
The last ( ) Function 108
The local-name ( ) Function 109
The name ( ) Function 111
The namespace-uri ( ) Function 112
The position ( ) Function 113
Number Functions 115
The ceiling ( ) Function 115
The floor ( ) Function 116
The number ( ) Function 117
The round ( ) Function 118
The sum ( ) Function 120
String Functions 120
The concat ( ) Function 120
The contains ( ) Function 122
The normalize-space ( ) Function 123
The starts-with ( ) Function 124
The string ( ) Function 126
The string-length ( ) Function 127
The substring ( ) Function 129
The substring-after ( ) Function 130
The substring-before ( ) Function 131
The translate ( ) Function 133
Boolean Functions 135
Looking Ahead 135

Chapter 3 XPath Data Model 137

XPath Data Model 137
The In-Memory Tree 138
The Source Tree 138
The Result Tree 144
Documents and Trees 145
Node Types 146
Root Node 148
Element Nodes 148
Attribute Nodes 149
Comment Nodes 150
Namespace Nodes 150
Processing Instruction Nodes 151
Text Nodes 151
Node Names 152
Document Object Model 152
DOM Level 3 XPath 153
DOM and XPath Object Models 153

Text Nodes 153
Namespace Nodes 154
Document Type Declaration 155
Conclusion 155

XML Information Set 155
Information Items 156
Document Information Item 157
Element Information Item 158
Attribute Information Item 159
Processing Instruction Information Item 160
Unexpanded Entity Reference Information Item 160
Character Information Item 161
Comment Information Item 161
Document Type Declaration Information Item 162
Unparsed Entity Information Item 162
Notation Information Item 163
Namespace Information Item 163

Perspective 163

Looking Ahead 164

Chapter 4 The Four XPath Syntaxes 165

Unabbreviated Absolute Syntax 166
The Child Axis 167
The Attribute Axis 169
The Descendant Axis 174
The Parent Axis 178
The Ancestor Axis 178
The Following-Sibling Axis 178
The Preceding-Sibling Axis 178
The Following Axis 179
The Preceding Axis 179
The Namespace Axis 179
The Descendant-or-Self Axis 179
The Ancestor-or-Self Axis 181
The Self Axis 181

Unabbreviated Relative Syntax 181
The Child Axis 183
The Attribute Axis 186
The Descendant Axis 187
The Ancestor Axis 190
The Following-Sibling Axis 193
The Preceding-Sibling Axis 196
The Following Axis 198
The Preceding Axis 199
The Namespace Axis 201

The Descendant-or-Self Axis 203
The Parent Axis 206
The Ancestor-or-Self Axis 207
The Self Axis 209

Abbreviated Syntax 210

Abbreviated Absolute Syntax 211
The Child Axis 211
The Attribute Axis 212
The Descendant Axis 212
The Parent Axis 214
The Ancestor Axis 214
The Following-Sibling Axis 214
The Preceding-Sibling Axis 215
The Following Axis 215
The Preceding Axis 215
The Namespace Axis 215
The Descendant-or-Self Axis 215
The Ancestor-or-Self Axis 216
The Self Axis 216

Abbreviated Relative Syntax 216
The Child Axis 216
The Attribute Axis 218
The Descendant Axis 219
The Parent Axis 219
The Ancestor Axis 220
The Following-Sibling Axis 220
The Preceding-Sibling Axis 220
The Following Axis 220
The Preceding Axis 221
The Namespace Axis 221
The Descendant-or-Self Axis 221
The Ancestor-or-Self Axis 221
The Self Axis 221

Looking Ahead 221

Chapter 5 XPath Functions 223

The Why of XPath Functions 223

Node Set Functions 224
count ( ) Function 224
id ( ) Function 230
last ( ) Function 231
local-name ( ) Function 232
name ( ) Function 234
namespace-uri ( ) Function 235

position ( ) Function 237
Number Functions 238
ceiling ( ) Function 238
floor ( ) Function 238
number ( ) Function 239
round ( ) Function 239
sum ( ) Function 239
String Functions 239
concat ( ) Function 239
contains ( ) Function 241
normalize-space ( ) Function 243
starts-with ( ) Function 244
string ( ) Function 244
string-length ( ) Function 244
substring ( ) Function 245
substring-after ( ) Function 245
substring-before ( ) Function 245
translate ( ) Function 245
Boolean Functions 245
boolean ( ) Function 245
false ( ) Function 246
lang ( ) Function 247
not ( ) Function 248
true ( ) Function 248
XSLT Functions 248
XSLT current ( ) Function 248
XSLT document ( ) Function 249
XSLT element-available ( ) Function 251
XSLT format-number ( ) Function 252
XSLT function-available ( ) Function 253
XSLT generate-id ( ) Function 253
XSLT key ( ) Function 256
XSLT system-property ( ) Function 259
XSLT unparsed-entity-uri ( ) Function 259
Looking Ahead 259

Chapter 6 Using XPath and XSLT to Produce XML 261

What XPath Can't Do 261
The <xsl:output> Element 262
Creating Elements 267
Selecting and Creating Elements 267
Using the <xsl:element> Element 267
Using the <xsl:copy> Element 270
Using the <xsl:copy-of> Element 273

Reordering Content 274
Reusing Business Information 277
Creating Attributes 280
Looking Ahead 282

Chapter 7 Using XPath and XSLT to Produce HTML 283

A Simple HTML Example 283
Creating an HTML List 285
Creating an HTML Table 293
Creating a Pseudo Schema in HTML 299
Looking Ahead 304

Chapter 8 Using XPath and XSLT to Produce SVG 305

Introduction to SVG 305
The SVG <line> Element 306
The SVG <rect> Element 308
The SVG <text> Element 309
SVG Tools 309
Creating a Static SVG Bar Chart 310
Creating an Animated SVG Bar Chart 319
Creating a Static SVG Line Chart 323
A Weather Chart in a Scrolling SVG Text Window 327
Limitations of Transformations to Produce SVG 331
Looking Ahead 332

Chapter 9 Using XPath in XPointer 333

What Is XPointer? 334
HTML Fragment Identifiers 334
What XPath Cannot Express 338
Understanding XPointer 339
XPointer Terminology 339
The XPointer Data Model 341
Point Locations 342
Range Locations 343
Character Escaping 344
XPointer's Three Syntaxes 344
Full XPointers 345
Bare Names 345
Child Sequences 345
XPointer's Two Schemes 347
The xpointer Scheme 347
The xmlns Schem 347

XPointer's Role 348
XPointer Functions 348
end-point ( ) Function 349
here ( ) Function 349
origin ( ) Function 349
range ( ) Function 349
range-inside ( ) Function 350
range-to ( ) Function 350
start-point ( ) Function 350
string-range ( ) Function 350
Further Development of XPointer 351
Looking Ahead 351

Chapter 10 XForms and XPath 353

Overview of XForms 354
Differences between HTML/XHTML Forms and XForms 355
How It Looks to the User 355
Separating Purpose and Presentation 357
Form Controls 358
The Textbox Form Control 359
The Secret Form Control 360
The uploadMedia Form Control 360
The selectOne Form Control 360
The selectMany Form Control 361
The selectBoolean Form Control 361
The Range Form Control 362
The Button Form Control 362
The Output Form Control 362
The Submit Form Control 362
The Reset Form Control 363
Common Child Elements 363
Moving from HTML to XForms 363
How XForms Works 366
XForms User Interface 366
User Interface—Dynamic Interface 366
User Interface—Repeating Items 367
User Interface—Interface Templates 368
User Interface—Layout 368
The XForms Model 368
Model Item Properties 369
The name Property 369
The type Property 369
The readOnly Property 369
The required Property 369

The relevant Property 369
The calculate Property 369
The priority Property 370
The validate Property 370
Using Datatypes in the XForms Model 370
XForms Terminology 371
XForms Elements 372
The xform Element 372
The model Element 373
The instance Element 373
The submitInfo Element 373
The bind Element 373
XForms Properties 373
The immediate-refresh Property 373
The immediate-revalidate Property 374
The immediate-recalculate Property 374
The use-nils Property 374
The Read-Only Properties 374
XForms Processors 374
The X-Smiles Browser 374
Mozquito XForms Preview Release 375
Cardiff.com LiquidOffice 375
XPath in XForms 375
Instance Data 375
XPath Context in XForms 376
Context for Outermost Binding Elements 376
Context for Non-outermost Binding Elements 377
Binding Expressions 377
Canonical Binding Expressions 378
Datatypes 378
XForms Masks and Regular Expressions 379
XForms-specific Datatypes 380
The Currency Datatype 380
The Monetary Datatype 380
Multiple Forms in Containing Document 380
XForms Function Library 381
Number Functions 381
The average ( ) Function 381
The min ( ) Function 381
The max ( ) Function 381
The count-non-empty ( ) Function 381
String Functions 382
The now ( ) Function 382
The xforms-property ( ) Function 382

Boolean Functions 382
The submit ( ) Function 382
The reset ( ) Function 382
Looking Ahead 382

Chapter 11 XPath in Canonical XML and XML Signatures 383

Overview of XML Security Specifications 383
Principles of Security 384
Canonical XML 386
Why Canonical XML Is Needed 386
What Does Canonical XML Do? 392
What Is canonicalization? 393
XPath, Subsets, and Canonical XML 394
Document Order 397
The Final Step 397
Document Subsets 399
Well-Formed 399
Some Canonicalization Examples 400
Exclusive XML Canonicalization 402
XML Signatures 402
Using Canonical XML with XML Signatures 404
XPath Transforms in XML Signatures 404
XPath Filtering in XML Signatures 404
Language-Specific Implementations 405
XACML and XPath 405
Looking Ahead 406

Chapter 12 Selecting Elements and Attributes 407

Selecting Elements 407
Selecting Elements by Name 408
Selecting Elements by Parent Characteristics 410
Selecting Elements by Value 410
Selecting and Sorting Elements by Value 412
Selecting Elements by Position 414
Selecting a Preceding Element 419
Selecting Following Elements 423
Selecting Elements by Attribute Presence 425
Selecting Elements by Attribute Value 430
Attribute Less than a Specified Value 433
Selecting Elements When Passing Parameters 435
Looking Ahead 440

Chapter 13 Working with XPath Functions 441

Using Node-Set Functions 441
Using the count ( ) Function 441
Using the id ( ) Function 446
Using the last ( ) Function 449
Using the local-name ( ) Function 451
Using the name ( ) Function 454
Using the namespace-uri ( ) Function 456
Using the position ( ) Function 458

Using Number Functions 460
Using the sum ( ) Function 460

Using String Functions 466
Using the concat ( ) Function 466
Using the contains ( ) Function 468
Using the starts-with ( ) Function 472

Using Boolean Functions 475
Using the boolean ( ) Function 475
Using the not ( ) Function 477

Looking Ahead 478

Chapter 14 XPath 2.0 and XQuery 479

XPath 2.0 Working Drafts 479
XPath 2.0 Requirements 480
Manipulation of XML-Schema Typed Content 480
Manipulation of String Content 482
Support Related XML Standards 482
Improve Ease of Use 482
Improve Interoperability 483
Improve Internationalization Support 483
Maintain Backward Compatibility 483
Enable Improved Processor Efficiency 483
XPath 2.0 Data Model 483
Data Typing in XPath 2.0 485
Accessors 485

The Need for XQuery 486

Keeping Up-to-Date 486

XSLT 2.0 487

Conclusion 487

Appendix Online Resources 489

World Wide Web Consortium 489

XPath Sites 490

FLAMFLY

XSLT Sites 490
SVG Sites 490
XForms Sites 491
XPointer Sites 491
XQuery Sites 491
XSL-FO Sites 492
General XML Sites 492

Glossary **493**

Index **501**

Introduction

Welcome to XPath Essentials. I sometimes think that XPath is one of the cinderella XML technologies: It is busy working away in the background, out of sight. If we didn't have XPath, then the work wouldn't get done. So, if you are working with XML, you almost certainly will need some understanding of XPath to get the work done.

As you will see as you work through this book, XPath is used already in XSLT (Extensible Stylesheet Language Transformations) and its use is emerging in other technologies such as XPointer, XForms, Canonical XML, XML Signatures, and the like.

Who This Book Is For

This book is a learn-by-doing book. It is for anybody who needs to start using XPath today. Throughout the chapters of this book, you will be shown example after example of working code that uses XPath. As you work through the text and the examples, I think you will find code that you can adapt or extend to your own needs and purposes. Because the primary use of XPath at the time of writing is with XSLT, many of the examples in this book are focused on transforming XML documents either to HTML/XHTML or to other XML formats.

If you have a background in HTML, XHTML, XML, or XSLT, that will help move you along faster. If you don't have any markup language background, don't worry. The introductory material in Chapter 1 will give you a start with XML and XSLT that will give you a framework for later reading. If you have problems later in this book you may want to refer back to Chapter 1 or its examples to review the necessary foundational knowledge.

My aim in presenting many examples is to get you on your own feet using XPath as quickly as possible.

How This Book is Organized

In Chapter 1 I have reviewed in a fairly compacted way foundational information on XML 1.0 itself, XML Namespaces, and XSLT. If you are new to XPath, you may find some of this a little dense, but don't worry too much if the relevance of some technical issues isn't immediately clear. You can refer back to these sections as you work through later chapters, if you find that you haven't the background knowledge at your fingertips.

Chapter 2 is an overview of the XML Path Language, XPath. A little of everything in the language is touched on as a basis for more detailed discussion and further examples later in the book.

Chapter 3 examines the XPath Data Model. XPath works on in-memory tree models rather than the markup tags you see in XML documents. I also briefly look at other XML models: the Document Object Model and the XML Information Set.

Chapter 4 looks at the four different forms of XPath syntax. For many simple uses, abbreviated syntax will meet your needs, but for some techniques you will also need to understand XPath location paths only expressible in the unabbreviated syntax.

Chapter 5 looks at XPath Functions and how they can be used to manipulate node sets, strings, and numbers.

Chapter 6 looks at using XPath with XSLT to create new XML documents.

Chapter 7 builds on many examples earlier in the book to explore the use of XPath with XSLT to create HTML output from XML source documents.

Chapter 8 examines how we can use XPath together with XSLT to create Scalable Vector Graphics, SVG, output files.

Chapter 9 looks at the extensions of XPath that form part of the XML Pointer Language, XPointer, the XML-based language for accessing fragments in XML documents.

Chapter 10 explores the use of XPath in the emerging W3C XForms standard.

Chapter 11 looks at the role XPath will play in XML-based security standards, in particular in the new Canonical XML and XML Signatures specifications.

Chapters 12 and 13 are review chapters, which take you through further examples of using XPath with XSLT, both in the selection of elements and attributes from source XML documents and further examples of usage of the XPath Functions.

Chapter 14 looks ahead to XPath 2.0 and its relationship with the XML Query Language, XQuery. Although XPath 2.0 is still some distance ahead, it will make several significant changes in XPath and open up a huge new potential usage of XPath with XQuery.

XPath Tools

These are described in Chapter 1 together with details of installation, as appropriate.

The Code for the Examples

The code for the examples is included in its entirety on the Wiley Web site: www.wiley.com/compbooks/watt.

Getting Started

It's time now to get into the meat of the book. Turn now to Chapter 1. Enjoy.

Acknowledgments

I would like to thank Jeni Tennison who performed an excellent job as tech editor, picking up those flaws that are so easy for an author to overlook.

I would also like to thank Scott Amerman and Penny Linskey of Wiley for their help in guiding this book through the process to completion.

XPath Essentials

TEAMFLY

Starting with XML and XSLT

The XML Path Language, XPath, is designed to allow the navigation of XML documents, often with the purpose of selecting individual elements, attributes, or some other part of an XML document for specific processing.

XPath was originally intended for use with, or as part of, two XML-based technologies: the Extensible Stylesheet Language Transformations (XSLT) and the XML Pointer Language (XPointer).

This chapter will review the basics of XML, with particular emphasis on aspects that are relevant for the understanding and use of XPath. If you are new to XPath, some of the examples may include terms with which you are not familiar. These will be explained in Chapter 2.

In addition to summarizing relevant parts of XML, this chapter will also review the basics of the Extensible Stylesheet Language Transformations, since many of the examples will demonstrate the use of XPath in the context of XSLT. Suitable tools for the editing of XML and for the processing of XSLT will also be discussed.

Overview of XML

The Extensible Markup Language (XML) is the context in which the XML Path Language, XPath, exists. XML provides a standard syntax for the markup of data and documents.

The first Recommendation for XML was issued by the World Wide Web Consortium (W3C) in February 1998 (located at www.w3.org/TR/1998/REC-xml-19980210). This XML 1.0 Recommendation was revised a little in another Recommendation termed the "Extensible Markup Language (XML) 1.0 (Second Edition)" Recommendation issued in October 2000. That revised Recommendation for XML 1.0 is located at www.w3.org/TR/2000/REC-xml-20001006.

NOTE If any revision of XML 1.0 Second Edition takes place during the lifetime of this book it will be located at www.w3.org/TR/REC-xml. If that URL displays the XML 1.0 (Second Edition) Recommendation of October 2000, you will know that there has been no further final revision of the XML Recommendation since this chapter was written.

XML is a meta-language, by which I mean it is a "language" designed for the creation of other application languages. The application languages of XML can cover an enormous variety of uses. Since these application languages all use XML syntax, we can use XPath to navigate around these documents and select parts of them for a variety of programming purposes.

Among the XML application languages that you might already have met is XSLT, the Extensible Stylesheet Language Transformations, which you will see frequently used in this book as the framework within which the use of XPath will be illustrated. Another example of an XML application language is the Scalable Vector Graphics language(SVG), which describes vector graphics using a syntax that is compliant with the XML 1.0 Recommendation.

Thus when we speak of something as being written "in XML," it is an imprecise use of language. What we mean is that the document is written in an application language of the XML meta-language and corresponds to the syntax standards laid out in the XML 1.0 Recommendation. The application language may be a formally defined complex language such as XSLT and SVG or it may be a very simple informal application language, destined perhaps never to be used again.

A Simple XML Document

A simple XML document may, but need not, contain an XML declaration as its first line. If an XML declaration is present, nothing must precede it, not even a single space character. The XML declaration may, but need not, contain an encoding attribute, which describes the character encoding of the document. All XML processors are required to support UTF-8 and UTF-16 character encoding.

Listing 1.1 is a simple example XML document with an XML declaration as its first line.

As you can see, an XML document is made up of elements contained in angled brackets, with a start tag, such as <BasicXMLFacts>, and a matching end tag, such as </BasicXMLFacts>. Within the <BasicXMLFacts> element is a number of <Fact> elements. An XML element may have attributes that describe the element in some way. Each <Fact> element in the simple example in Listing 1.1 has a type attribute, and in each case the

```
<?xml version='1.0'?>
<BasicXMLFacts>
<Fact type="Simple”>XML is a meta-language</Fact>
<Fact type="Simple">XSLT is an application language of XML</Fact>
<Fact type="Simple">XPointer is a fragment identifier language of
  XML</Fact>
<Fact type="Simple">XPath is not written in XML syntax</Fact>
</BasicXMLFacts>
```

Listing 1.1 A Simple XML Document (BasicXMLFacts.xml).

value of the type attribute happens to be “Simple”. In addition to a type attribute, each <Fact> element also contains some text content.

We can use XPath in conjunction with an XSLT stylesheet to output the information contained in that document as HTML for display. A stylesheet to create HTML output is shown in Listing 1.2.

The DisplayFacts named XSLT template inserts an HTML ordered list within the HTML skeleton created in the “main template.” The DisplayFacts template, using the XSLT <xsl:for-each> element, takes the content of each <Fact> element and creates an HTML list item to contain it. The output from that transformation is shown in Figure 1.1.

```
<?xml version='1.0'?>
<xsl:stylesheet version=”1.0”
                xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”>

<xsl:template match=”/”>
<html>
<head>
<title>Some XML Facts</title>
</head>
<body>
<h3>Some Facts about XML and XPath</h3>
<xsl:call-template name=”DisplayFacts”/>
</body>
</html>
</xsl:template>

<xsl:template name=”DisplayFacts”>
<ol>
<xsl:for-each select=”/BasicXMLFacts/Fact”>
<li><xsl:value-of select=”.”/></li>
</xsl:for-each>
</ol>
</xsl:template>
</xsl:stylesheet>
```

Listing 1.2 A Stylesheet to Display Information in BasicXMLFacts.xml as HTML (BasicXMLFacts.xsl).

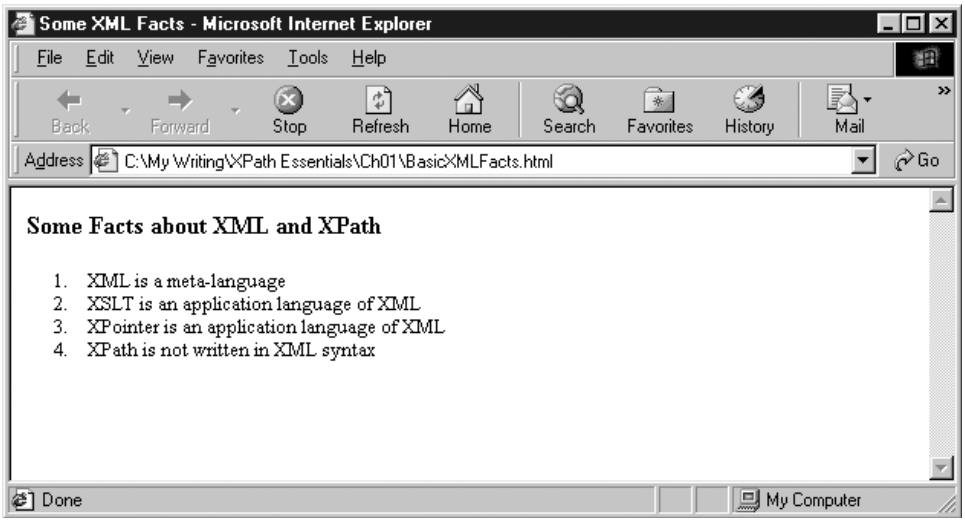

Some XML Facts - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch01\BasicXMLFacts.html
Go
Some Facts about XML and XPath
1. XML is a meta-language
2. XSLT is an application language of XML
3. XPointer is an application language of XML
4. XPath is not written in XML syntax
Done
My Computer

Figure 1.1 The Content of the XML Document Displayed as HTML.

Let's move on and explore XML in more detail.

Structure of an XML Document

As you saw in the simple example above, an XML document—even a very simple one—has a structure.

An XML document, in one sense, is simply a series of characters. What makes it an identifiable XML document is the fact that those characters are employed in such a way as to describe a logical structure. Since XPath operates on that logical structure, it is pretty important that we have a good grasp of it.

Logical

The logical structure of an XML document is determined in large part by the elements it contains. The attributes present on those elements provide additional information about the logical structure of the document.

All XML documents must be "well-formed," a term which we will discuss in more detail later.

Elements

All XML documents contain one or more elements. If an element contains content, whether other elements or text, then it must have a start tag and an end tag. In the following code

```
<AnElementWithContent>Some content goes here.</AnElementWithContent>
```

the start tag is

```
<AnElementWithContent>
```

the end tag is

```
</AnElementWithContent>
```

The text contained between the start tag and the end tag is the element's content.

An element may have no content and is then said to be an *empty element*. Typically an empty element would contain useful information in one or more attributes. An empty element may be written in either of two ways. It may be written either with a start tag and an end tag, like this:

```
<AnEmptyElement></AnEmptyElement>
```

or by using an abbreviated syntax like this:

```
<AnEmptyElement/>
```

Both forms mean the same. If you use the form with the start tag and end tag, you need to be careful not to allow any content, not even a space character, since this could be expected to generate an error when the element was processed by a validating XML parser.

Types

The name in the element's start and end tags gives the element's type. The name is often referred to as the *element type name*. An alternative term for the element type name is *generic identifier*.

The element types in an XML document may be constrained by element declarations, in the internal or external subset of a Document Type Definition (DTD). In a DTD no element type may be declared more than once.

An element declaration may describe the following permitted content models:

- Element content (a sequence of one or more child elements)
- Mixed content (character data optionally interspersed with child elements)
- EMPTY
- ANY

A schema expressed in XML, such the W3C's XSD Schema (also, ambiguously called XML Schema), is an alternate way to constrain the content of an XML document. XPath 1.0, the topic of this book, does not make use of XSD Schema but future versions of XPath are likely to use it.

Attribute Specification

An element may have one or more attributes, which typically provide additional information about the element type or its content. An attribute, if present in the XML document, is written within the start tag of an element:

```
<AnElement anAttribute="an attribute value"></AnElement>
```

In the above line of code the text "anAttribute" is termed the attribute name. The attribute name is followed by an equals sign. The text contained within quotes (which may be paired single or double quotes) is termed the attribute value.

When an element has more than one attribute

```
<AnElement secondAttribute="something" firstAttribute="something
else"></AnElement>
```

the ordering of the attributes is not significant.

The name='value' pairs are referred to as an element's attribute specification.

NOTE The delimiters of attribute name-value pairs may be either single quotes or double quotes. Those delimiters must be used in pairs, as in attribute='value' or attribute="value".

The attribute types in an XML document may be constrained by attribute-list declarations appropriately linked to the element declarations of the elements that contain the attributes, either in the internal or external subset of a Document Type Definition. Attribute-list declarations specify the name, data type, and default value (if any) of each attribute associated with a given element type.

Attribute-list declarations may be used to:

- Define the set of attributes allowed for an element type
- Constrain the type of the attributes
- Provide default values for the attributes

Attributes may be:

- String types
- Tokens (ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS)
- Enumerated types (a notation or an enumeration)

An XPath processor will typically operate after any default attributes are incorporated into the source XML document. If they are defined in the external subset of a DTD, which cannot be accessed, unexpected results may occur, due to the absence of expected attributes.

For example, if we had in an XML document a <Paragraph> element that had a type attribute of value "warning" and which was supplied from a DTD

```
<Paragraph type="warning">
```

we might use an XPath expression to select for <Paragraph> elements that possessed such an attribute, with code like this:

```
Chapter/Caution/Paragraph[attribute::type="warning"]
```

If, due to some technical problem, the DTD could not be accessed, then use of the XPath expression would fail to select the desired element nodes.

Physical

An XML document, although it is logically one, may be composed of several storage units. The XML term for such a storage unit is an *entity*. Each entity has content and, with the exception of the document entity and the external subset of the Document Type Definition, each entity has a name.

Entities may be of two types: parsed and unparsed. The content of a parsed entity is referred to as its replacement text. The content of an unparsed entity may or may not be text, and if the content is text then it may or may not be XML. Each unparsed entity has an associated notation, which is identified by name. XML places no restrictions on unparsed entities, other than that the XML processor must make available the identifier and notation for the unparsed entity to a containing application.

Parsed entities are invoked by name using entity references. Unparsed entities are accessed by name, which are given in the values of ENTITY or ENTITIES attributes.

Document Entity

Each XML document has one, and only one, document entity. The document entity functions as the starting point for an XML processor and when an XML document is contained in a single file, then the document entity contains the whole document. The document entity has no name.

In XPath the processing of the hierarchical tree-like version of an XML document begins at the root node, which provides the point of reference for an XPath processor, relative to where all other nodes are located.

Broadly, the XPath root node corresponds to the document entity in the source XML document.

Character and Entity References

A *character reference* refers to a specific character in the ISO/IEC 10646 character set. Unfortunately the International Organization for Standardization (ISO) does not make its standards available online free of charge in the way that the W3C does. ISO has a Web site, located at www.iso.ch, where some general information can be found.

For each entity reference an Entity Declaration must exist, except that the entities &, ', >, <, and " do not need to be declared (see Predefined Entities section, which follows).

An entity is said to be "included" when its replacement text is retrieved, processed, and inserted into the document at the location where the entity reference exists. The replacement text may contain character data and (except for parameter entities) markup, which must be processed in the normal way.

In order to validate an XML document, a validating XML processor must include the replacement text of a parsed entity when it recognizes one. If a nonvalidating parser

processes a document, then it must inform the application that an external entity has been recognized but not included. This approach allows browsing of an XML document to take place, with the option of marking the existence of the external entity and for its expansion on user request.

Entity Declarations

An entity may be declared in the internal subset or the external subset of the Document Type Definition.

An entity declaration takes the form

```
<!ENTITY name "replacement text">
```

The name identifies the entity in an entity reference or, in the case of an unparsed entity, it identifies the value of an ENTITY or ENTITIES attribute.

If we wanted to create an entity with the name XPathSlogan, we could do so using the following entity declaration:

```
<!ENTITY XPathSlogan "XPath helps an XML processor get around.">
```

And wherever we used an entity reference in our XML document to &XPathSlogan;, it would be replaced at the appropriate place in the document with the text "XPath helps an XML processor get around."

If the entity declaration declares an entity value, as above, then the entity is termed an internal entity.

An external entity is declared like this:

```
<!ENTITY MyFile SYSTEM "c:\My Files\AFile.xml">
```

The name of the external entity is "MyFile" and its replacement text is to be found in a file located on the system of the computer which contains the XML document at the location c:\My Files\AFile.xml.

Parsed Entities

An external parsed entity should begin with the *text declaration*:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

The text declaration is very similar to the XML declaration but it does not have a standalone attribute, which, of course, can be present on an XML declaration. If a text declaration is present, it must occur at the beginning of an external parsed entity.

A well-formed XML document is required to have a single element root. An external parsed entity is not required to satisfy that requirement, since when it is expanded it will be nested within an XML document, which already has a single element root.

An important consequence of well-formed entities is that parts of an XML document, including elements, processing instructions, comments, and entities, may not start in one entity and finish in another.

An external parsed entity need not use the same character encoding as the containing XML document.

Parameter Entities

Parameter entities are similar to character entities in that they can substitute for a defined string. A parameter entity is used to provide a shortcut for markup declarations and parts of declarations. Therefore parameter entities can only be used in DTDs.

All parameter entities are well-formed by definition.

Predefined Entities

XML and, by implication, XML processors provide or recognize five predefined entities. They are:

- & (the ampersand character, &)
- ' (the apostrophe character, ', also known as the single quote character)
- > (the greater than character, >, also used as the final character in the start tag or end tag of an element)
- < (the less than character, <, also used as the first character in the start tag or end tag of an element)
- " (the quote character, ", also known as the double quote character)

To avoid an error, it may be necessary to replace an occurrence of these five characters within parsed character data. The presence, for example, of a "<" character as in

```
4 < 5
```

might be interpreted by an XML processor as the beginning of a new tag before the previous one had been completed, which would be an error.

An alternative approach to using the predefined entities is to use character references, such as < for the "<" character and & for the "&" character to escape these characters when they occur in character data.

Parsed Character Data

XML entities contain data that can be parsed or unparsed (typically non-XML data). The character data is composed of characters, some of which are markup and some of which are text.

Except when it is contained in a CDATA section, character data is parsed by the XML processor and certain characters are not permitted since they may be confused with the first, "<", and last, ">", characters of a start tag or end tag. Such characters must either be "escaped"—using one of the predefined entities just described as a character reference—or contained within a CDATA section.

Notations

Notations identify by name the format of unparsed entities, the format of elements that possess a NOTATION attribute, or the application to which a processing instruction is targeted.

Notation declarations provide a name for the notation so that it can be used in entity and attribute declarations. In addition, an external identifier is provided for the notation that can provide a way for an XML processor or the application running it to locate a helper application, which may be able to process the information in the notation in question.

Syntax

An XML document may consist of several components, each of which is permitted or defined in the XML 1.0 Recommendation.

XML Declaration and the Prolog

An XML document may optionally begin with an XML declaration, which specifies which version of XML is being used. When an XML declaration is present, it must be the first thing in the XML document. It cannot be preceded by even a single space character without causing an error.

The XML declaration, if present, must have a version attribute. Currently the only relevant value for the version attribute is "1.0." In addition the XML declaration may optionally include encoding and standalone attributes. If the document has external declarations, then the standalone attribute should have the value "no."

The Document Type Declaration contains and/or points to declarations of the elements, attributes, and so on, contained in the XML document. Such declarations provide a grammar for the class of documents. The grammar is contained in a Document Type Definition, which potentially has two parts: an internal subset and an external subset.

NOTE The Document Type Declaration and the Document Type Definition are not the same, although they are closely related. The Document Type Declaration contains the internal subset, if it exists, of the Document Type Definition. The Document Type Declaration also contains a reference to the external subset, if it exists, of the Document Type Definition.

If we had a document that had an element root called <MyElement> then the Document Type Declaration might look like this:

```
<!DOCTYPE MyElement [internal subset]>
```

If the Document Type Definition has an internal subset, it is contained within the square brackets in the Document Type Declaration, as shown above.

If the Document Type Definition also has an external subset, then the syntax is as follows:

```
<!DOCTYPE MyElement MyDTD.dtd [internal subset]>
```

In addition to the XML declaration and the Document Type Declaration, the Prolog of an XML document may contain comments, processing instructions, and whitespace. Following the Prolog, the element root of the document occurs.

Comments

The syntax for XML comments is the same as in HTML.

```
<!-- This is a comment. -->
```

A comment begins with "<!—" and ends with "—>". A comment may not contain the string "—" except as part of the terminating sequence of the comment.

Processing Instructions

Processing instructions provide a means for XML documents to contain instructions for applications which may operate on them. The syntax of a processing instruction is:

```
<?target instructions ?>
```

The target of the processing instruction is the application that is intended to respond to the processing instruction. The "instructions" part consists of an arbitrary string, which is implicitly expected to be formulated in such a way as to make sense to the target application.

Probably the most commonly used processing instruction takes this form:

```
<?xml-stylesheet href="AnXSLTStyleSheet.xsl"
  type="text/xsl" ?>
```

This form may be included in the prolog of an XML document and associates that XML source document with an XSLT stylesheet named AnXSLTStyleSheet.xsl, which is located in the same directory as the XML source document. The type attribute indicates the media type of the referenced stylesheet. For XSLT stylesheets the media type is "text/xsl".

Elements

The way to write elements was defined earlier in the chapter. An element may have one or several attributes, or it may have none.

If a Document Type Definition exists, the permitted content of each element is declared there.

CDATA Sections

A CDATA section may occur anywhere character data may occur. A CDATA section provides a way to include significant volumes of text without the need to escape multiple characters within it.

If I were creating this book in an XML structured document and wished to refer to some XML code, such as

```
<Greeting>Hello World!</Greeting>
```

I would need to escape with < and >, as appropriate, at each occurrence of the less than and greater than characters. With code of that length, it may be less convenient to create a CDATA section like this:

```
<![CDATA[<Greeting>Hello World!</Greeting>]]>
```

But with multiple or lengthy quotes describing markup, it becomes much more convenient to use a CDATA section to escape disallowed characters in lengthy portions of text.

Language Identification

It is sometimes useful to know what natural language a document is written in. XML provides the xml:lang attribute to fulfill that purpose. The value taken by an xml:lang attribute is a two-character language code, derived from the IETF RFC 1766 (see www.ietf.org/rfc/rfc1766.txt).

If we wished to define the content of a particular <paragraph> element as being in English, we could do so like this:

```
<Paragraph xml:lang="en">This paragraph is written in
  English.</Paragraph>
```

Well-formed and Valid

XML documents can be classified by two criteria: whether or not they are well-formed and whether or not they are valid.

Well-Formed

Strictly speaking all XML documents are well-formed. By that I mean that the XML 1.0 Recommendation indicates that only when a document that looks like XML is well-formed does it actually merit the term "XML." In practice, it is probably more realistic to talk of XML documents that are well-formed and not well-formed.

For an XML document to be well-formed it needs to satisfy a number of criteria set out in the XML 1.0 Recommendation, including those in the following list:

- The element type name in the end tag must match the element type name in the start tag.
- No attribute name may appear more than once in any start tag or empty element tag.
- Attribute values may not contain direct or indirect entity references to external entities.
- An attribute value may not contain the "<" character, nor may the replacement text for an entity contain a "<".
- An entity reference must not contain the name of an unparsed entity.
- A parsed entity must not contain a recursive reference to itself.

A full description of the criteria for well-formed is found in the XML 1.0 Recommendation.

Valid and Non-Valid

In addition to being well-formed, XML documents may be examined to determine whether or not they are valid according to a schema. In this context a schema may be a Document Type Definition (DTD), which is expressed in Extended Backus Naur Format

(EBNF) as described in the XML 1.0 Recommendation. Or it can be a schema that is itself expressed in XML such as the XSD Schema (also known as "XML Schema"), RELAX NG (the successor to the RELAX and TREX schema language proposals), or a number of other XML schema types written in XML.

For an element to be valid according to a DTD, it must satisfy the following criteria, in addition to the requirements for being well-formed as already mentioned. For each element there must be a corresponding element declaration (either in the internal subset of the Document Type Declaration, or in its external subset). In addition, one of the following must be true:

- If the declaration matches EMPTY the element must have no content.
- If the element is declared to have child elements, the content of the element matches what is declared, and the content of those child elements also matches the declared content for them.
- If the element content is declared to be Mixed, then the content of the element consists of character data plus child elements whose element type names match elements declared in the content model.
- If the declaration matches ANY, then the types of child elements have been declared.
- No element may have more than one ID attribute.
- An ID attribute must have a declared value of #IMPLIED or #REQUIRED.
- No element type may have more than one notation attribute declared.

If an attribute is present on an element, then for it to be valid it must have been declared on that element and the value of the attribute must be of the type declared for it.

In an element declaration where the content is of mixed type, the same name may not appear more than once.

XSD Schema, also known as W3C XML Schema, provides additional constraints on the structure of XML documents beyond the limited range provided by a DTD and provides much improved datatyping facilities compared to the rudimentary typing provided by a DTD. XSD Schema is the subject of an upcoming book in the XML Essentials Series, titled *XML Schema Essentials*.

Normalization

As you will see later, XPath expressions and location paths frequently occur within the values of XML attributes. For example, when selecting an XPath node using the XSLT <xsl:value-of> element, we would typically find an XPath expression as the value of that attribute as shown here:

```
<xsl:value-of select='attribute::type'/>
```

Before being processed, the value of all attributes are "normalized." The process starts with the empty string and characters are added to that string, depending on each character in the value of the attribute.

For example, all whitespace characters (#x20, #xD, #xA, #x9) are converted to single space characters (#x20). For a character reference, the referenced character is inserted

in the position occupied by the character reference. For a normal character, that character is appended to the end of the normalized value.

XML Uses

The nature of tasks for which XML is perceived to be suited has undergone a significant shift and expansion since 1998. In part this is due to significant shifts during the same period in the perception of how the World Wide Web may be used.

When it was under development, XML was viewed as "SGML for the Web." The Standard Generalized Markup Language (SGML) is, like XML, a meta-language from which other application languages can be created. SGML is focused primarily on document-centric uses and is used by government departments and major corporations to provide the structural context for major documentation projects, for example, the documentation of a commercial airliner, its safety testing, and its maintenance needs.

One application language of SGML, which was already widespread on the World Wide Web, was the HyperText Markup Language (HTML). One of the problems with HTML was that the structure of the document had become increasingly blurred with aspects of HTML designed to describe how a Web document was to be presented by a Web browser. Such integration of structure and presentation is convenient and efficient when small documents or small Web sites are being created but causes significant difficulties, including incurring major maintenance costs, when a Web site is large and when maintenance is needed. For example, if a Web site with 2000 pages is to be given a new style and color, with HTML used alone each of the 2000 pages may have to be edited individually to achieve the desired change in corporate online livery.

If structure and presentation are separated, then the maintenance task is eased. An improvement in that direction can be achieved by using HTML with Cascading Style Sheets (CSS). However, HTML is limited in that it is a language with a fixed number of defined elements, which are limited in the way they can describe the content of a document.

XML is therefore well suited to the storage or transfer of data without mixing large amounts of presentation-oriented information into the data. The flexibility of XML would allow the inclusion of such presentation information, but then the benefits of the separation of content and presentation are lost.

The data stored as XML (or in a database with built-in facilities to convert to XML) can be transformed into HTML/XHTML for presentation directly on the Web (such transformations are described in Chapter 7, Using XPath and XSLT to Produce HTML). At the present time this is possibly the major use of XPath and XSLT.

Alternatively, XML may be converted from one structure to another using XPath and XSLT. To take a simple example, a publisher and a bookseller may both hold information about books in XML but use slightly different structures to store the same information. For example, a publisher might use an element-oriented structure shown in Listing 1.3.

A bookseller might choose to use attributes rather than elements, as shown in Listing 1.4.

To convert from the format used by the publisher (Listing 1.3) to the format used by the bookseller (Listing 1.4), we can use a stylesheet such as the one shown in Listing 1.5.

The output from that transformation is shown in Figure 1.2.

```
<?xml version='1.0'?>
<BookCatalog1>
<Book>
<Title>XPath Essentials</Title>
<Author>Andrew Watt</Author>
<PublnYear>2001</PublnYear>
<Publisher>Wiley</Publisher>
</Book>
</BookCatalog1>
```

Listing 1.3 Book Catalog Version 1 (BookCatalog1.xml).

```
<?xml version='1.0'?>
<BookCatalog2>
<Book Title="XPath Essentials" Author="Andrew Watt">
<PublnYear>2001</PublnYear>
<Publisher>Wiley</Publisher>
</Book>
</BookCatalog2>
```

Listing 1.4 Book Catalog Version 2 (BookCatalog2.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<xsl:apply-templates select="BookCatalog1"/>
</xsl:template>

<xsl:template match="BookCatalog1">
<xsl:element name="BookCatalog2">
<xsl:apply-templates select="Book"/>
</xsl:element>
</xsl:template>

<xsl:template match="Book">
<xsl:element name="Book">
<xsl:attribute name="Title"><xsl:value-of
```

Listing 1.5 A Stylesheet to Convert from One Book Catalog Structure to the Other (BookCatalog1.xsl). (*continues*)

```
select="Title"/></xsl:attribute>
<xsl:attribute name="Author"><xsl:value-of
select="Author"/></xsl:attribute>
<xsl:copy-of select="PublnYear"/>
<xsl:copy-of select="Publisher"/>
</xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Listing 1.5 (*Continued*)

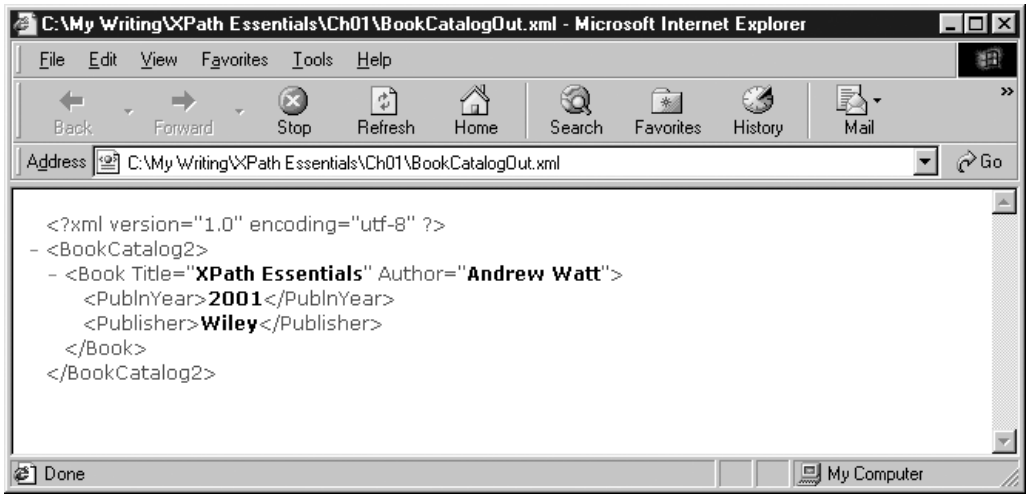


Figure 1.2 The Output of the Transformation from Book Catalog 1 to Book Catalog 2.

Of course, production documents are much more complex than these simple examples. A corporation may deal with multiple customers and suppliers. While there may be industrywide standards for structuring certain types of documents in XML, a corporation may deal with other companies in several sectors. The ability to use XML, XPath, and XSLT to implement the multiple necessary data structure conversions is immensely important for efficient conduct of many modern businesses.

In the first couple of years of XML, these were the major uses of XPath and XSLT, but as additional pieces of the XML jigsaw progressively emerge from W3C, the use of XPath is beginning to expand and can be expected to expand even more in the years to come.

The XML Technologies Jigsaw

Before examining the likely growth areas for use of XPath, let's first take an overview of what has been emerging from W3C in the XML track.

In the time since the release of the XML 1.0 Recommendation in February 1998, the number of XML-related specifications either released by the World Wide Web Consortium or under ongoing development is such that it is difficult for anyone to keep fully up to speed with all the details. By coincidence, as I was writing this chapter, a first working draft of a new XML-based specification for XQueryX, the XML Query Language dialect expressed in XML, has appeared (see www.w3.org/TR/xqueryx for additional information).

The following are the full XML-related Recommendations already issued by W3C, together with the relevant URLs, which you can follow up if you are interested:

- Canonical XML (www.w3.org/TR/xml-c14n)
- Document Object Model (DOM) (www.w3.org/DOM/)
- Mathematical Markup Language (MathML) (www.w3.org/Math/)
- Namespaces in XML (www.w3.org/TR/REC-xml-names)
- Resource Description Framework (RDF) (www.w3.org/RDF/)
- Scalable Vector Graphics (SVG) (www.w3.org/Graphics/SVG/)
- Synchronized Multimedia Integration Language (SMIL) (www.w3.org/AudioVideo/)
- XHTML (www.w3.org/MarkUp/)
- XML Path Language (XPath) (www.w3.org/TR/xpath)
- XML Schema (www.w3.org/XML/Schema)
- XSLT, Extensible Stylesheet Language Transformations (www.w3.org/TR/xslt)

Other specifications are under active development, some of which will be close to or have reached Recommendation status by the time you read this, while others are early in their development. Among the significant specifications currently under active development are

- XForms (www.w3.org/MarkUp/Forms/)
- XML Base (www.w3.org/TR/xmlbase/)
- XML Encryption (www.w3.org/Encryption/2001/)
- XML Information Set (www.w3.org/TR/xml-infoset)
- XML Linking Language (XLink) (www.w3.org/XML/Linking)
- XML Pointer Language (XPointer) (www.w3.org/XML/Linking)
- XML Protocol (www.w3.org/2000/xp/)
- XML Query (www.w3.org/XML/Query)
- XML Signature (www.w3.org/Signature/)
- XSL, Extensible Stylesheet Language Formatting Objects (www.w3.org/Style/XSL/)

It isn't my purpose here to summarize each of these specifications. Each reader will have an individual perspective on which specifications are relevant. The use of XPath fits into what is already a complex multidimensional XML jigsaw. As those specifica-

tions currently under development reach completion, the range and potential complexity of uses of XML, and therefore of XPath, will likely grow exponentially.

As indicated earlier XPath is mainly currently used in or with XSLT. XPath was designed to be used with XPointer too but the development of XPointer has been subject to many delays and, at the time of writing, it has not yet reached Recommendation status at W3C.

Collection of data on the Web currently often relies on HTML-based forms. W3C has an XML-based forms specification, XForms, under development which significantly extends the functionality provided by HTML forms. XForms uses XPath at its heart, as we will discuss in Chapter 10, XForms and XPath.

As the volume of data that is held as XML has increased, the importance of controlling access to sensitive data or querying XML data has become apparent. Security-oriented specifications such as Canonical XML and XML Signatures use XPath, as does the XML Query Language, XQuery, currently in the early stages of development at W3C. These uses of XPath will be considered in Chapter 11, XPath in Canonical XML and XML Signatures, and in Chapter 14, XPath 2.0 and XQuery.

Putting XML to Work

Constructing a simple XML document is straightforward. For example, to convey a welcome to the world of XPath we might create a simple XML document such as in Listing 1.6.

It is straightforward (and legal) in XML to create new elements whose element type name can convey something about the structure or meaning of the element. It is far easier for you to understand the simple purpose of the example in Listing 1.6 than to decipher any meaning from an HTML document like the one shown in Listing 1.7.

To a human reader the HTML is almost as easily understood as the XML, but for a computer the <h1> and <p> tags convey no meaning at all, other than how the content is to be presented on screen.

At present Web browsers have limited capability to display XML and, therefore, as mentioned earlier, data is often held as XML to be transformed to HTML for display. In due time XML browsers will become widely available, so that there will be much greater scope for creating an XML-based Web, both at the level of data storage, as now, and additionally using XML application languages for presentation, data collection, and so on.

```
<?xml version='1.0'?>
<SimpleDocument>
<Greeting>
Welcome to the world of XPath, the XML Path Language!
</Greeting>
<Context>
XPath Essentials
</Context>
</SimpleDocument>
```

Listing 1.6 A Simple Welcome to XPath (SimpleDocument.xml).

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>An XPath greeting</TITLE>
</HEAD>
<BODY>
<h1>Welcome to the world of XPath, the XML Path Language!</h1>
<P>
XPath Essentials
</P>
</BODY>
</HTML>
```

Listing 1.7 A Simple Welcome to XPath in HTML (SimpleDocument.html).

XML Namespaces

If XML were a rarely used syntax there might not be a pressing need for XML namespaces. However, since it is becoming increasingly a routine occurrence for an XML document to contain elements from more than one application language of XML, it is important that we, or the various types of applications that process XML, can reliably distinguish elements that might have the same name. The Namespaces in XML Recommendation is designed to let both a human reader and an application reliably distinguish between elements from different namespaces.

An XML namespace is a collection of names of element type names and attribute names which is identified by a URI reference.

Namespaces from a namespace may be displayed as a Qualified Name, sometimes called a *QName*. A QName consists of three parts: a namespace prefix, a colon character, and a local part. MyPrefix:MyElement is a QName, with the namespace prefix being "MyPrefix" and the local part being "MyElement".

NOTE **A namespace prefix must be what the Recommendation calls a NCName, that is, an XML name which excludes the colon character. XML names are permitted, by the XML 1.0 Recommendation, to include a colon character. But if the namespace prefix could contain a colon character, it would be difficult for a processor to distinguish such a colon character within the name from the colon character that separates the namespace prefix from the local part.**

If we wanted to use that QName on an element in an XML document that recognized the constraints of the Namespaces in XML Recommendation, we need to have a namespace declaration on the <MyPrefix:MyElement> element or an ancestor of it. To do that we would use the following code:

```
<MyPrefix:MyElement
xmlns:MyPrefix="http://www.AWonderfullyLongAndAwkwardName.com/Schemas/">
<!-- Element content goes here. -->
</MyPrefix:MyElement>
```

There are two reasons for having the apparently cumbersome approach of a namespace prefix, a namespace declaration, and a namespace URI. First, prefixes are (or should be) relatively short and thus human friendly, whereas a namespace URI can be very long and awkward to use as a prefix. Secondly, some characters can occur in URIs that are not permitted in XML names. By using a prefix that uses only characters allowed in XML names and associating that prefix with a URI, which may contain disallowed characters, the demands of XML are met while allowing the use of the full gamut of namespace URIs.

NOTE The namespace declaration uses one of two possible reserved attribute names—xmlns or an attribute name with xmlns: as its first part.

Attribute names are either namespace declarations of the type just described or are QNames. If the namespace declaration uses the form

```
xmlns=" http://www.AWonderfullyLongAndAwkwardName.com/Schemas/">
```

then the namespace declared is said to be the default namespace. In that situation no namespace prefix is used and the colon character preceding the local part of the QName is omitted.

Let's suppose we had a file of author names held as XML as shown in Listing 1.8.

We could additionally hold a file of books as shown in Listing 1.9.

Notice that both types of XML documents include a <Title> element. If we bring these documents together, how can we be sure to correctly distinguish the title of the book from the title of the person? Namespaces in XML provide a solution.

The first step is to declare a namespace, which is identified by means of a Uniform Resource Identifier (URI), and associate that namespace URI with a prefix, which is used in conjunction with the local part of the element type name, separated from the local part by a colon.

We can modify the start tag of the <Authors> element so it looks like this:

```
<?xml version='1.0'?>
<Authors>
<Author>
 <Title>Mr.</Title>
 <FirstName>Andrew</FirstName>
 <LastName>Watt</LastName>
</Author>
</Authors>
```

Listing 1.8 A Brief Authors Listing in XML (Authors.xml).

```
<?xml version='1.0'?>
<Books>
<Book>
<Title>XPath Essentials</Title>
<Publisher>Wiley</Publisher>
</Book>
<!-- Other books would be listed here. -->
</Books>
```

Listing 1.9 A Brief Books Listing in XML (Books.xml).

```
<au:Authors xmlns:au="http://WileyAuthors.com/Schemas">
```

Notice that we now have an <au:Authors> start tag. The "au" is called the namespace prefix. It is a human-friendly indication of the namespace. The code

```
xmlns:au="http://WileyAuthors.com/Schemas"
```

is called the namespace declaration, and it associates the namespace prefix, "au" in this case, with the (fictional) namespace URI, "http://WileyAuthors.com/Schemas". An XML processor uses the namespace URI to uniquely, or so it is hoped, identify an element. If document authors use only URIs over which they have legitimate rights, then correct and unique namespace usage should follow.

Our authors listing document, when revised to include namespace information, will look like Listing 1.10.

If we also add a namespace declaration to the books file, as shown in Listing 1.11, there is little likelihood of a human reader confusing the <au:Title> element with the <bks:Title> element. Similarly, a namespace-aware XML processor will recognize the respective namespace URIs of the two types of <Title> elements and will reliably distinguish them in any combined document.

XPath is namespace-aware. In XPath namespaces are represented as namespace nodes associated with particular element nodes. The concepts of nodes and the characteristics of namespace nodes and element nodes will be explained in Chapter 2.

```
<?xml version='1.0'?>
<au:Authors xmlns:au="http://WileyAuthors.com/Schemas">
<au:Author>
 <au:Title>Mr.</au:Title>
 <au:FirstName>Andrew</au:FirstName>
 <au:LastName>Watt</au:LastName>
</au:Author>
</au:Authors>
```

Listing 1.10 The Authors Listing Incorporating XML Namespaces (AuthorsNS.xml).

```
<?xml version='1.0'?>
<bks:Books xmlns:bks="http://WileyComputerBooks.com/Schemas/">
<bks:Book>
<bks:Title>XPath Essentials</bks:Title>
<bks:Publisher>Wiley</bks:Publisher>
</bks:Book>
<!-- Other books would be listed here. -->
</bks:Books>
```

Listing 1.11 The Books Listing Incorporating XML Namespaces (BooksNS.xml).

Where Does XPath Fit?

One answer, which I mentioned earlier, is found in the Abstract of the XPath Recommendation, which states, "XPath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer." At the time that the XPath 1.0 Recommendation was written, no draft existed of the XML Query Language; therefore the potentially immensely important synergy between XPath and XML Query could not be expressed.

At the present time the use of XPath with XSLT is the dominant one. That is not surprising since XSLT, like XPath, has been a full W3C Recommendation since November 1999, whereas XPointer and XML Query are both at Working Draft status at the time of writing.

Unfortunately, it has taken considerable time for the development of the XML Pointer Language, XPointer, with which XPath was designed to work,. XPath was released as a full Recommendation in November 1999, and the XPointer specification in August 2001 was still only at Working Draft status, with considerable controversy regarding its future development and few tools providing any worthwhile functionality.

The use of XPath with XPointer is described in Chapter 9, "Using XPath in XPointer."

What Can XPath Do?

XPath makes it possible for a processor to navigate around the hierarchy of XPath nodes to a particular part or parts of the document and may return a set of such nodes, called a node set, or may return a value which is a string, a number, or a Boolean value (i.e., true or false).

A particularly important use of XPath is for matching. For example, if you wanted to transform an XML source document and present selected elements of it as HTML, then an XPath expression can be written to do that. The XPath expression would allow those selected elements (and only those elements), or, more precisely, the nodes which represent those elements, to be matched by the templates or elements in the XSLT stylesheet and their content displayed suitably within the resulting HTML document.

What Are the Next Steps?

If you are to get the full benefit from this book to equip you to understand and use XPath, then you need certain basic knowledge of XML (as described earlier in this chapter), an understanding of XSLT (to be described later in the chapter), and some XML and XSLT tools at hand.

Knowledge Needed

You will need at least a basic understanding of XML. Earlier in this chapter I briefly summarized some salient points regarding XML, but if your knowledge or experience of XML is limited, I suggest that you take a look at an introductory book on XML. Once we get past this chapter, I will assume that you have a basic grasp of XML.

Most of the examples in this book that make use of XPath are written in the context of XSLT stylesheets. I won't assume that you have a detailed knowledge of XSLT, but I will assume a basic knowledge. I will briefly summarize some salient points about XSLT later in this chapter, but, depending on your current level of knowledge of XSLT, you may well find it helpful to consult *XSL Essentials* in this series or a similar book on XSLT as you go along to fill in any blanks in your knowledge of XSLT.

And lastly you need software to edit XML documents and XSLT stylesheets and to process the XML documents using XSLT stylesheets. After discussing the software you will need, I will summarize salient points about XSLT that you will need to understand as you read the rest of the book.

Software

There are now a huge number of different pieces of XML-oriented software available and quite likely you will already have found some software that allows you to write and edit XML and to carry out XSLT transformations. If so, then please feel free to use those since the pieces of software I am about to recommend have many competitors, which are also good. Two widely used XML editors that you might want to consider are XML Writer and XML Spy.

XML Writer, currently at version 1.2.1, is a straightforward XML editor with color coding of XML. It is relatively inexpensive, about $50, and does a straightforward, unobtrusive job as an XML editor. It allows checking of XML documents for "well-formedness" and also can validate XML documents against a DTD. You can create template documents that you can reuse, which saves a lot of time when creating many similar documents.

NOTE **XML Writer will transform XML documents using XSLT. Be careful that you have MSXML3 (or later) installed if this is to work correctly. See the examples on www.xmlwriter.com to test capabilities on your machine. See the later section in this chapter about where to find and how to install MSXML3.**

XML Spy (currently at version 3.5) is a more sophisticated and complex XML editor than XML Writer, but the full version costs about three times as much. It provides

several views of your data, for example, as text or as a hierarchy, which you can expand or collapse as desired as you are working. It also provides an autocomplete function that can be useful. Although not directly relevant to XPath, XML Spy can automatically create an XML Schema from an XML instance document.

Where to Get Help

The definitive and authoritative source for all information on XML and the related technologies is the World Wide Web Consortium Web site at www.w3.org.

The XPath 1.0 Recommendation is located at www.w3.org/TR/xpath. The XSLT 1.0 Recommendation is located at www.w3.org/TR/xslt.

There is a very useful and active XSL mailing list based at www.mulberrytech.com. Despite the name, the XSL mailing list, in the recent past at least, focuses much more on XSLT than XSL Formatting Objects (XSL-FO) and also has a useful smattering of questions and discussion about XPath.

Of course XPath will be potentially useful too with XSLT and XSL-FO. There is an XSL-FO mailing list on YahooGroups.com at www.yahoogroups.com/group/XSL-FO.

There is a mailing list dedicated to support of the Saxon XSLT processor (mentioned later in this chapter), which you may find helpful if you are using Saxon or Instant Saxon. Further information about the Saxon mailing list is available at www.saxon.xsl.listbot.com/.

Introduction to XSLT

Most of the examples that use XPath in this book will apply XPath within XSLT stylesheets. Thus, it is essential, if you are to fully grasp the uses of XPath, that you have some grasp of XSLT. If you are already familiar with and/or have some experience using XSLT, feel free to skim or skip this brief introduction to XSLT.

XSLT, the Extensible Stylesheet Language Transformations, is a declarative language which uses *stylesheets* (sometimes called *transformation sheets*) to describe a transformation of an XML source document to a result document, which may be a differently structured XML document, an HTML document, a plain text document, or some other format. An XSLT transformation does not change or transform the source document—it simply creates a new result document. In the rules in an XSLT stylesheet, you declare what you want the XSLT processor to output. It is, generally speaking, up to the XSLT processor how it goes about processing a source document to achieve the specified output.

More precisely, it is not a source "document" that an XSLT processor processes. It actually processes a source *tree* held in memory and creates an output tree in memory based on the nodes contained in the source tree and the template rules contained in the XSLT stylesheet. The source tree is the in-memory node-based hierarchical representation of a source XML document. The output tree may, but need not, be later saved as a file—in XML, HTML, or some other format—or may be processed further without ever having been *serialized* (that is, changed back into conventional XML syntax). In the ex-

amples in this book I will typically save the output of XSLT transformations in a file in serialized form, since that is the form easiest to discuss and demonstrate.

An XSLT stylesheet is a well-formed XML document and it also conforms to the Namespaces in XML Recommendation, which was discussed earlier in this chapter. The elements to be transformed by an XSLT stylesheet are selected using XPath location paths (or expressions) perhaps in combination with XPath or XSLT functions (which are introduced in Chapter 2, XPath Fundamentals, and described in more detail in Chapter 5, XPath Functions).

An XSLT stylesheet takes the basic form shown in Listing 1.12.

The XML declaration is optional as in all XML documents. The <xsl:stylesheet> element is the element root of an XSLT stylesheet. There is an alternative to the <xsl:stylesheet> element as element root—the <xsl:transform> element. Listing 1.13 has identical semantics to the code shown in Listing 1.12.

An XSLT processor processes an XSLT stylesheet identically whether it has an <xsl:stylesheet> or <xsl:transform> element as element root.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<!-- The meat of the transformation takes place here and/or other
templates are referred
to from here. -->
</xsl:template>
</xsl:stylesheet>
```

Listing 1.12 A Skeleton XSLT Stylesheet (SkeletonXSLT.xsl).

```
<?xml version='1.0'?>
<xsl:transform version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<!-- The meat of the transformation takes place here and/or other
templates are referred
to from here. -->
</xsl:template>
</xsl:transform>
```

Listing 1.13 A Skeleton XSLT Stylesheet Using the <xsl:transform> Element (SkeletonTransform.xsl).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head></head>

<body>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Listing 1.14 An XSLT Stylesheet Incorporating the Skeleton for an HTML Output Document (SkeletonXSLTHTML.xsl).

An XSLT processor will begin processing a source tree—the in-memory hierarchical representation of your source XML document—at the <xsl:template> element which has a match attribute with the value of "/". In other words the XSLT processor starts processing at the template which matches the root node of the source tree.

Within the <xsl:template> element in the following code, we specify literal output which creates a bare skeleton of an HTML document, as shown in Listing 1.14.

Elements that are not in the XSLT namespace, such as the HTML elements shown in Listing 1.14, are output literally. Elements from the XSLT namespace such as the <xsl:value-of> element shown in Listing 1.15 would be further processed by the XSLT processor and the content specified by the select attribute of the <xsl:value-of> element would be output, not the XSLT element itself.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head></head>
<body>
<h2><xsl:value-of select="/Text"/></h2>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Listing 1.15 Literal Output and XSLT Elements to Be Processed (SkeletonXSLTProcess.xsl).

```
<?xml version='1.0'?>
<Text>
Welcome to the world of XPath and XSLT!
</Text>
```

Listing 1.16 A Text Welcome in XML (TextWelcome.xml).

The value of the select attribute of the <xsl:value-of> element is an XPath location path, in this case meaning select the content of the node that represents the <Text> element which is a child of the root node. Thus, if Listing 1.16 was our source XML document, and we applied to it the XSLT stylesheet just described, we would produce an HTML document like the one shown in Listing 1.17.

The output code shown is what is produced by the Instant Saxon XSLT processor, but tidied up a little for presentation. The output of the XSLT transformation displays in a browser is as shown in Figure 1.3.

The <xsl:template> element is one of the elements which may be direct children of the <xsl:stylesheet> element. There are eleven other so-called top-level elements, although since they are contained within the <xsl:stylesheet> element they might have been better called second-level elements. These elements provide control, for example, over the import or inclusion of multiple XSLT stylesheet modules and define how the output tree is to be constructed. For example, using the <xsl:output> element it is possible to define whether output will be XML, HTML, or text.

The code within an <xsl:template> element causes nodes to be written to the output tree. The types of nodes written to the output tree are determined by the code within the <xsl:template> element and any other <xsl:template> elements that it calls.

The facility to call one XSLT template from another permits modular design of code, which has the well-known advantages in terms of writing, testing, debugging, and division of labor among members of a software development team.

```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
</head>
<body>
<h2>Welcome to the world of XPath and XSLT!</h2>
</body>
</html>
```

Listing 1.17 A Welcome to XPath Transformed into HTML (TextWelcome.html).

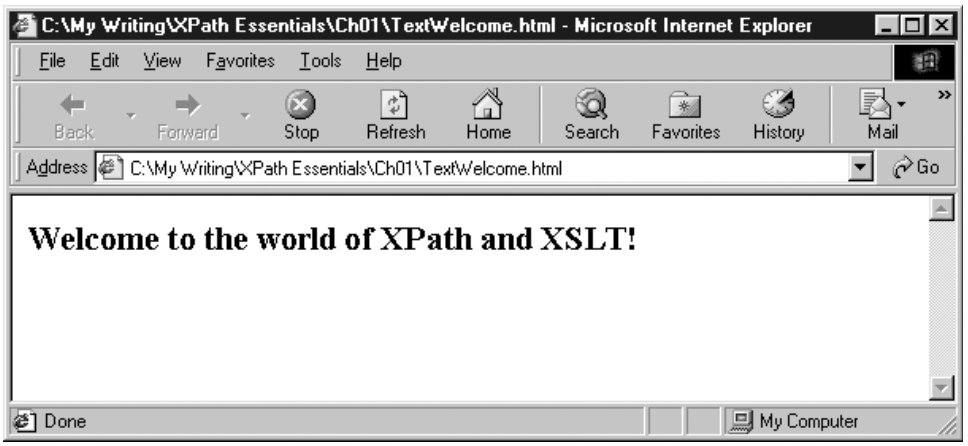

C:\My Writing\XPath Essentials\Ch01\TextWelcome.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch01\TextWelcome.html
Go
Welcome to the world of XPath and XSLT!
Done
My Computer

Figure 1.3 The Appearance of TextWelcome.html in Internet Explorer 5.5.

XPath in XSLT

In this general process of using XSLT to create an output tree, XPath fulfills the crucial role of enabling you to choose which elements, attributes, or other parts of the source tree you show to a user in any particular context.

XPath expressions and location paths are used to select nodes for inclusion in the result tree. If an XPath location path matches the node you want to output, then the desired output is created.

XPath expressions can be used, in effect, to hide information from some or all users. If, for example, you had selling details of the price you paid suppliers within the information that you held about a company's product, you would not want your customers to be aware of that. Such sensitive or confidential information could be selectively suppressed from being part of the output by omitting those elements from the XPath expressions used to create the output tree.

The <xsl:stylesheet> Element

As mentioned earlier the <xsl:stylesheet> element, or its synonym the <xsl:transform> element, forms the skeleton of all XSLT stylesheets. In this section we will look a little more closely at its permitted attributes.

NOTE **Strictly speaking, the <xsl:stylesheet> element is optional in that there is a form of stylesheet called the literal-result-element-as-stylesheet form where no <xsl:stylesheet> element is used. This is described in Chapter 2.2 of the XSLT Recommendation. All XSLT examples in this book will use the full form of syntax for an XSLT stylesheet.**

An <xsl:stylesheet> element is the element root both of full stylesheets and of stylesheets or stylesheet modules that are imported or included by means of the use of the <xsl:import> or <xsl:include> elements (to be described shortly).

The <xsl:stylesheet> element may take four attributes. One, the version attribute, is required. The other three—the id attribute, the extension-element-prefixes attribute, and the exclude-result-prefixes attribute—are optional.

At the present time the version attribute may only take the value of "1.0." Once future versions of XSLT are approved by W3C then alternative values are likely to be permitted.

NOTE If you see references online to XSLT 1.1 you should be aware that W3C has abandoned development of XSLT 1.1 and has instead begun the possibly lengthy process of developing an XSLT 2.0 specification.

The id attribute is optional. It would be of use if an XSLT stylesheet were embedded within another XML document. In that case the id attribute would provide a way to refer to the XSLT stylesheet by name.

The extension-elements-prefixes attribute is used when extension elements are present in a stylesheet. Each such extension element must be prefixed by a namespace prefix that is not in the XSLT namespace.

The exclude-result-prefixes attribute designates namespace prefixes present in the stylesheet that are to be excluded from the result document, unless they are actually used in the result document.

In addition to the four attributes mentioned above, the <xsl:stylesheet> element requires a namespace declaration for the XSLT namespace of the form

```
xmlns:xsl="http://www.w3.org/XSL/1999/Transform"
```

which identifies the XML document as an XSLT 1.0 stylesheet.

Of course the chosen namespace prefix need not be "xsl" but that is the convention, even though it would make more sense to use a prefix of "xslt" since the W3C is using the term "XSL" to refer primarily to XSL-FO.

NOTE A different namespace declaration was used for the now obsolete Microsoft "XSL" dialect, of the form xmlns:xsl="http://www.w3.org/TR/WD-xsl". There are sufficient differences between that old draft dialect and the W3C XSLT Recommendation that I suggest you avoid it if possible. Microsoft has moved to support W3C XSLT as of MSXML3.

XSLT Top-Level Elements

There is a group of XSLT elements which occur only as direct child elements of the <xsl:stylesheet> element. These elements are called the *top-level elements*. They control or influence many general aspects of the behavior of an XSLT transformation.

There are twelve XSLT top-level elements. An <xsl:import> element (or elements) if present must precede any other top-level elements present in a stylesheet. With that exception, the ordering of top-level elements within a stylesheet is flexible.

NOTE **You may come across the mention of another element—the <xsl:script> element—as a top-level element. In XSLT 1.0, it is not a top-level element. There are two contexts in which you might be introduced to <xsl:script>—as part of the now obsolete Microsoft "XSL" or in the context of "XSLT 1.1," whose development has now been abandoned by the W3C (see www.w3.org/TR/2001/WD-xslt11-20010824/).**

All XSLT stylesheets which you see will contain, explicitly or implicitly, an <xsl:template> element. For the purposes of this book, all XSLT stylesheets will be shown with explicit <xsl:template> elements for clarity.

The <xsl:import> Element

The <xsl:import> element is chief among the top-level elements, in that if it is present in a stylesheet it needs to precede any other top-level elements. The <xsl:import> element causes the stylesheet at the location specified in its href attribute to be imported. Thus, to import the ToBeImported.xsl stylesheet we could use <xsl:import> like this (assuming that ToBeImported.xsl was situated in the same directory as the importing stylesheet):

```
<xsl:import href="ToBeImported.xsl"/>
```

The value of the href attribute may be a relative URI, as shown in the example above, or an absolute URI. If the URI is relative then it is interpreted in the light of the base URI of the importing stylesheet.

The file reference in the value of the href attribute must be a valid XSLT stylesheet. The top-level <xsl:stylesheet> element of the imported stylesheet is, in effect, discarded and the top-level children of the discarded element are inserted into the importing stylesheet in place of the <xsl:import> element. However, the base URI for imported elements remains that of the imported stylesheet, rather than the importing one, which has relevance, if the base URIs of the importing and imported stylesheets differ when using any relative URIs within the imported stylesheet.

The <xsl:attribute-set> Element

The <xsl:attribute-set> element is used to delineate a named set of attributes and values. Having a named set of attributes allows them to be defined in a single place and applied as a group to any output element.

The <xsl:decimal-format> Element

The <xsl:decimal-format> element is used in conjunction with the XSLT format-number ( ) function. When numbers are converted into strings using that function, the <xsl:decimal-format> element defines the characters and symbols to be used.

The <xsl:decimal-format> element is always an empty element. It has eleven optional attributes: name, decimal-separator, grouping-separator, infinity, minus-sign, NaN, per-cent, per-mille, zero-digit, digit, and pattern-separator.

The <xsl:include> Element

The <xsl:include> element provides a means to carry out textual inclusion, analogous to the process in other programming languages such as C. The <xsl:include> element has only one attribute, the href attribute which is a required attribute. Thus if we want to include a module called ModuleToBeIncluded.xsl we could do so using the following code, provided that ModuleToBeIncluded.xsl is in the same directory as the stylesheet within which the <xsl:include> element exists:

```
<xsl:include href="ModuleToBeIncluded.xsl"/>
```

The difference between <xsl:import> and <xsl:include> is that with <xsl:include> the semantics of the included stylesheet are not changed by the process, whereas an imported module will be wholly or partly overridden by similarly named templates in the importing stylesheet if such templates are present.

The <xsl:key> element

The <xsl:key> element is used in conjunction with XSLT key( ) function. The <xsl:key> element has three attributes—name, match, and use—all of which are required. The name attribute is a QName, which is the name of the key. The match attribute is a pattern, which defines which nodes the key applies to. The use attribute contains an expression used to arrive at the value of the key for each of the nodes selected, using the match attribute.

The <xsl:key> element(s) in a stylesheet are processed before any global variables. Thus the value of a global variable may make use of the value of a key, but not the other way around.

It may be helpful to see a brief example. Let's suppose we wanted to use a key to refer to some AMD CPUs. Our source XML document might include the following:

```
<CPUs>
<CPU type="AMDK6" MHz="400"/>
<CPU type="Duron" MHz="700"/>
<CPU type="Athlon" MHz="1300"/>
</CPUs>
```

In a stylesheet we might create a key for application to each CPU type as follows

```
<xsl:key name="AMDCPU" match="CPU" use="@type"/>
```

The value of the element name <CPU> is matched by the value of the match attribute in the key. The use attribute of the <xsl:key> element specifies that the value of the type attribute of the <CPU> element is used as the key.

The <xsl:namespace-alias> Element

The <xsl:namespace-alias> element is used to map the namespace URI used in the stylesheet to a different namespace URI in the output document. An important use of this element is when an XSLT stylesheet is used to create another XSLT stylesheet as

output. If the desired output elements were expressed directly in the XSLT namespace in the stylesheet during processing, an XSLT processor would treat these as XSLT elements to be processed rather than as elements that are to be output.

The <xsl:namespace-alias> is always an empty element and has two attributes (both are required). The two attributes are the stylesheet-prefix attribute and the result-prefix attribute. The former refers to the namespace prefix expressed on an element in the stylesheet that is being processed, and the latter refers to the namespace prefix on the same element when it is included in the output document. Both attributes must have a value, which is an NCName. An NCName is any legal XML name except that the colon character is not permitted. The reason for that is obvious—an XSLT processor would have difficulty distinguishing a colon character within a prefix from the colon character which separates a namespace prefix from the local part of the QName.

Thus if you wanted to use the following `<JustForNow:template>` as an indication that an `<xsl:template>` element should be output as part of an output XSLT stylesheet, then within the first XSLT stylesheet the <xsl:namespace-alias> element would look like this:

```
<xsl:namespace-alias
  stylesheet-prefix="JustForNow"
  result-prefix="xsl"/>
```

Assuming that the namespace prefix "xsl" for the output document was associated with the XSLT namespace URI,

```
xmlns:xsl="http://www.w3.org/XSL/1999/Transform"
```

in the output document we would have created an <xsl:template> element in the output document, without risking that the element is inadvertently processed during instantiation of the first XSLT stylesheet.

The <xsl:output> Element

The <xsl:output> element is used to determine the output format from an XSLT transformation.

Conceptually, during an XSLT transformation a result tree is created first and then that result tree is serialized to form, for example, an XML document. The <xsl:output> element operates on the serialization process, not the creation of the result tree.

NOTE **It is not a requirement of the XSLT Recommendation that an XSLT processor serialize the result tree—it can make it available for processing by, for example, some other API. In that case an <xsl:output> element can be expected to be ignored.**

The <xsl:output> element has ten attributes, all of which are optional.

The method attribute typically takes one of three values: xml, html, or text. Thus if you want to specify unambiguously that the output of a transformation is to be HTML you would use

```
<xsl:output method="html"/>
```

perhaps with some of the other permitted attributes. The default output is XML; therefore, when no <xsl:output> element is present the output is XML just as if the following were specified:

```
<xsl:output method="xml"/>
```

An exception to that rule occurs when the element root of the output document is <html>, in which case the XSLT processor will likely treat the output as HTML (if no method attribute is specified).

The version attribute specifies the version of the chosen output method that is to be implemented. Thus to output HTML 4.0 you could specify

```
<xsl:output method="html" version="4.0"/>
```

XSLT processors need not support all versions of possible outputs.

The encoding attribute specifies which character encoding scheme is to be used in the output document.

The omit-xml-declaration attribute applies when the output method is "xml." It may take the values of "yes" or "no." When the value is "no" an XML declaration is automatically produced at the beginning of a result document.

If an XML declaration is to be output, the standalone attribute of the <xsl:output> element indicates, if it is present, that a standalone attribute is to be included in the XML declaration. The standalone attribute on the <xsl:output> element has permitted values of "yes" and "no." The value specified is also the value in the standalone attribute in the XML declaration of the output document.

The doctype-public and doctype-system attributes of the <xsl:output> element each take a string value. That string value is the value to be output in the result document.

The cdata-section-elements attribute names those elements in the source document whose content is to be output in the result document as CDATA sections.

The indent attribute of the <xsl:output> element defines whether or not the output should be indented to reflect the hierarchical structure of nested elements. The attribute may take the values of "yes" and "no."

The media-type attribute indicates the media type to be associated with the output file. The media type is also known as the MIME type.

NOTE **An XSLT stylesheet may contain more than one <xsl:output> element. In that case it is as if the elements present are merged into one composite <xsl:output> element. If there is an apparent conflict of values of attributes, then explicitly declared values take precedence over default values and values with higher import precedence take precedence over those of lower precedence.**

The <xsl:param> Element

The <xsl:param> element may be used as a top-level element in which case it defines a global parameter. Alternatively, it may be used within an <xsl:template> element, when

it must come before any other child elements of the <xsl:template> element, to define a local parameter.

The <xsl:param> element has a required name attribute and an optional select attribute. The select attribute provides a default value for the parameter, if no explicit value is provided in the call. If an <xsl:param> element has a select attribute it should be an empty element; otherwise it may optionally contain a template body.

The way in which explicit values are supplied for global parameters varies between implementations. For local parameters an explicit value can be supplied using <xsl:with-param> as a child of an <xsl:apply-templates> or <xsl:call-template> element.

The <xsl:preserve-space> Element

The <xsl:preserve-space> element is used to control how whitespace nodes in the source XML document are handled. See also the <xsl:strip-space> element.

The <xsl:preserve-space> element has a required elements attribute. The value of the elements attribute is a whitespace separated list of names that defines which elements in the source XML document are to have their whitespace-only nodes preserved during processing.

Whitespace in XML consists of tab, newline, carriage-return, and space characters.

The <xsl:strip-space> Element

The <xsl:strip-space> element is used to control how whitespace nodes in the source XML document are processed. Its effects may be modified by the use of the <xsl:preserve-space> element.

The <xsl:strip-space> element is always empty and has a required elements attribute, the value of which is a whitespace separated list of names that defines which elements in the source XML document are to have their whitespace-only nodes stripped during processing.

The <xsl:template> Element

The <xsl:template> element defines a template that is used to create all, or, more usually, part of the result tree. An <xsl:template> element is instantiated either by matching nodes against a pattern or calling a template by name.

Matching against a pattern makes use of the match attribute of the <xsl:template> element, as when the root node is matched like this

```
<xsl:template match="/">
```

The pattern, which is the value of a match attribute, may not include a variable reference, so that circular references are avoided, since an <xsl:variable> element may have an <xsl:apply-templates> element child.

Similarly if you had a template whose start tag looked like this

```
<xsl:template name="GeorgeDubya">
```

then you could access it by using the following

```
<xsl:call-template name="GeorgeDubya"/>
```

The name attribute on the <xsl:call-template> element and the name attribute on the <xsl:template> element need to match exactly.

An <xsl:template> element must have either a match attribute or a name attribute, or both. When an <xsl:template> element has both a match attribute and a name attribute, it can be called by either an <xsl:apply-templates> element or an <xsl:call-template> element.

In addition to its match attribute and name attribute, both of which are optional, an <xsl:template> element may optionally have a priority attribute and a mode attribute.

The priority attribute has a positive or negative numerical value, which is used by the XSLT processor should there be several templates that match the same node.

The mode attribute is used to refine a set of nodes for processing. Only those nodes that possess a mode attribute with the desired value are considered for processing.

The <xsl:variable> Element

An <xsl:variable> element is used to declare a global or local variable in a stylesheet. When used as a top-level element, the <xsl:variable> element declares a global variable. When nested within an <xsl:template> element, an <xsl:variable> element declares a local variable.

The <xsl:variable> element has a required name attribute and an optional select attribute. The value of the name attribute is used elsewhere in an XSLT stylesheet to reference the value of the <xsl:variable> element. For example, if we declared an <xsl:variable> element like this

```
<xsl:variable name="CurrentPrice" select="//Stocks/MyStock/CurrentPrice"/>
```

then we could reference that variable from elsewhere in the stylesheet by using the notation `$CurrentPrice`, which is the value of the name attribute preceded by a $ character.

When a select attribute is present as in the example just seen, the XPath expression, which is the value of the attribute, is evaluated to give the value of the variable. If a select attribute is present, then the <xsl:variable> element should be an empty element. If an <xsl:variable> element has no select attribute, then the value of the variable is determined from the content of the <xsl:variable> element.

For example, to declare a specific literal string value as the value of a variable you could use this code:

```
<xsl:variable name="Capital">
Washington
</xsl:variable>
```

which is equivalent to either of the following lines of code:

```
<xsl:variable name="Capital" select="'Washington'"/>
```

or

```
<xsl:variable name="Capital" select='"Washington"'/>
```

NOTE **Unlike variables in many other programming languages, XSLT variables cannot be updated. There is no assignment statement in XSLT.**

Other XSLT Elements

In addition to the <xsl:stylesheet> element and the top-level elements, XSLT provides several other elements that occur within an <xsl:template> element or another top-level element.

NOTE **The <xsl:document> element is not an XSLT 1.0 element although it was proposed for inclusion in the now-abandoned XSLT 1.1.**

The <xsl:apply-imports> Element

The <xsl:apply-imports> element is used with imported stylesheets and hence the <xsl:import> element. The <xsl:apply-imports> element is an instruction and is therefore always used within a template. It has no attributes.

If a template exists in the imported module and a template of the same name exists in the importing stylesheet, then the imported module will be overridden. If, however, the template in the importing stylesheet uses the <xsl:apply-imports> element to make use of the imported template, then instead of a crude overriding of the imported template or module, a more graded modification of it can be made.

In the functionality of the <xsl:apply-imports> element you may recognize similarities to the relationship between super class and sub-class in object-oriented programming languages with the sub-class overriding methods in the super class.

The <xsl:apply-templates> Element

The <xsl:apply-templates> element defines a node-set to be processed. The <xsl:apply-templates> element is an instruction that is always used within a template. It selects a set of nodes in the input tree and causes each of them to be processed in turn by finding a matching template. If the <xsl:apply-templates> element has a nested <xsl:sort> element nested within it, then the <xsl:sort> element determines the order in which the nodes are processed. In the absence of an <xsl:sort> element the nodes are processed in document order. An <xsl:with-param> element may be used nested within an <xsl:apply-templates> element.

The <xsl:apply-templates> element can have two attributes—a select attribute, which defines what nodes are to be processed by the XSLT processor, and an optional mode attribute, which must match the mode attribute of a template if selected nodes are to be processed. Thus an <xsl:apply-templates> element takes the following form:

```
<xsl:apply-templates select="Expression" mode="QName"/>
```

The above <xsl:apply-templates> element would cause the following template to be instantiated:

```
<xsl:template match="Expression" mode="QName">
<!-- Template body goes here. -->
</xsl:template>
```

but not the following one, since it lacks a matching mode attribute.

```
<xsl:template match="Expression">
<!-- Template body goes here. -->
</xsl:template>
```

The following code would cause the templates that match a <Chapter> element, which is the child of a <Document> element, to be instantiated

```
<xsl:apply-templates select="/Document/Chapter"/>
```

In the absence of a select attribute, or if the select attribute has a value of "*", the code

```
<xsl:apply-templates/>
```

will cause the child nodes of the context node to be processed.

The <xsl:attribute> Element

An <xsl:attribute> element is used to cause an attribute node to be added to the result tree. Such an attribute node can be output only in certain conditions. A corresponding element node must have been written to the result tree and no node type other than an attribute node may have been added to the result tree before the attribute node is added. In practice this means that an <xsl:attribute> element is used in conjunction with either an <xsl:element> element or an <xsl:copy> element or with a literal result element.

So, for example, if you wanted to output an HTML element and wanted to add an align attribute to it you could use code that uses a literal result element:

```
<div>
<xsl:attribute name="align">left</xsl:attribute>
</div>
```

Alternatively, you could use the <xsl:element> element like this:

```
<xsl:element name="SharePrice">
<xsl:attribute name="type">DowJones</xsl:attribute>
</xsl:element>
```

An <xsl:attribute> element may occur within an <xsl:template> element or within an <xsl:attribute-set> element.

An <xsl:attribute> element has two attributes. The name attribute is required, may take the form of an attribute value template, and is the name in the name-value pair of

the attribute to be output. The namespace attribute is optional, may take the form of an attribute value template that returns a URI, and defines the namespace URI of the generated attribute.

NOTE **Attribute value templates are discussed later in this section. They take the form of an expression contained in curly braces, such as {MyExpression}. Thus the naming of an attribute could have code of this form: name="{MyExpression}".**

The value of the generated attribute is determined by the content of the <xsl:attribute> element. The value must be a string value.

You cannot use the <xsl:attribute> element to add namespace declarations to the output document by setting the name attribute to "xmlns" or "xmlns:SomeName". Namespace declarations are added automatically by the XSLT processor on elements that need them.

The <xsl:call-template> Element

I mentioned earlier that the <xsl:template> element may have a name attribute and that such a template can be called by name. The <xsl:call-template> element is used to do that. The <xsl:call-template> element has only one attribute, the name attribute, which is a required attribute. The value of the name attribute is a QName. The name attribute of the <xsl:call-template> element and the name attribute of the <xsl:template> element must match. If the name attribute is a QName, then it is not the namespace prefixes which are compared—it is the namespace URIs. For the template to be instantiated, both the namespace URIs and the local part of the QName must match.

The value of the name attribute must be written literally. There is no way, at least in XSLT 1.0, to use a variable to choose from a selection of named templates based on some runtime criterion. To make such a choice you would need to use an <xsl:choose> element and nest a selection of <xsl:call-template> elements within it.

Nested within an <xsl:call-template> element may optionally be one or more <xsl:with-param> elements.

If you had a template like this

```
<xsl:template name="Special">
<!-- The works of the template goes here. -->
</xsl:template>
```

then you can call that template using code like this

```
<xsl:call-template name="Special"/>
```

Or if you also wish to pass a parameter, it would take this general form:

```
<xsl:call-template name="Special">
<xsl:with-param name="SpecialParameter" select="SomeExpression"/>
</xsl:call-template>
```

For the parameter has been passed to be evaluated, there would need to be a corresponding <xsl:param> element within the <xsl:template> element, like this

```
<xsl:template name="Special">
<xsl:param name="SpecialParameter"/>
<!--The rest of the works of the template goes here. -->
</xsl:template>
```

The <xsl:call-template> element can be used recursively to process a list, either a node set or a list of separated strings. However, detailed consideration of that is beyond the scope of this summary.

The <xsl:choose> Element

An <xsl:choose> element is used to make a choice among a number of options. The <xsl:choose> element has no attributes. Its content is one or more <xsl:when> elements and an optional <xsl:otherwise> element.

Thus the <xsl:choose> element can, when it has one <xsl:when> child and one <xsl:otherwise> element child, be used similarly to an `if ... then ... else` type of statement in other programming languages.

```
<xsl:choose>
<xsl:when test="SomeExpression">
<!-- Do first thing -->
</xsl:when>
<xsl:otherwise>
<!-- Do something else -->
</xsl:otherwise>
</xsl:choose>
```

Or the <xsl:choose> element can be used similarly to a switch or case statement:

```
<xsl:choose>
<xsl:when test="SomeExpression">
<!-- Do first thing -->
</xsl:when>
<xsl:when test="SomeOtherExpression">
<!-- Do a second thing -->
</xsl:when>
<xsl:otherwise>
<!-- Do something else -->
</xsl:otherwise>
</xsl:choose>
```

Note that within an <xsl:choose> element, one option at most is ever processed. The first <xsl:when> element whose test attribute evaluates to "true" is processed, and all other <xsl:when> elements (whether or not their test attribute would evaluate to "true") and the <xsl:otherwise> element, if present, are ignored. If none of the <xsl:when> elements has a test attribute which evaluates to "true" then the <xsl:otherwise> element, if present, is instantiated.

Thus an <xsl:choose> element can be used to make a choice between, for example, several <xsl:call-template> elements, depending on the value of one or more test expressions. However, at most, one <xsl:call-template> element can be instantiated in this way.

NOTE An <xsl:choose> element with one <xsl:when> element child and no <xsl:otherwise> element child is equivalent to an <xsl:if> element.

The <xsl:comment> Element

The <xsl:comment> element is used to write a comment to the result tree and hence to the output document. An <xsl:comment> element has no attributes. Its content is written to the output document.

Thus, the following

```
<xsl:comment>
This section is about XPath functions and their use.
</xsl:comment>
```

would cause the following comment to be produced in the output document:

```
<!-- This section is about XPath functions and their use.-->
```

The <xsl:copy> Element

The <xsl:copy> element copies the current node in the XML source document to the result tree. Only the node itself is copied, not its attributes nor its child nodes or descendant nodes. This is a so-called shallow copy. To achieve a deep copy, use the <xsl:copy-of> element (see later in this section).

The <xsl:copy> element has one optional attribute: the use-attribute-sets attribute.

The effect of using the <xsl:copy> element varies, depending on the type of node that is the current node when the <xsl:copy> element is instantiated.

If the current node is an element node, then it is as if the <xsl:element> element were used with the name attribute set to the element type name of the current element node. Any namespace nodes associated with the element node are also copied. If a use-attribute-sets attribute is present, then the sets of attributes specified in that attribute are added to the copy of the element node in the result tree. If the <xsl:copy> element has a template body as content then that template body is instantiated.

If the current node is an attribute node, then it is as if the <xsl:attribute> element was used with the name attribute set to the name of the current attribute node. If there is no suitable element node in the result tree then an error occurs. Any use-attribute-sets attribute or template body is ignored when the current node is an attribute node.

If the current node is a namespace node, then a new namespace node is created in the result tree, with the same name and value as the current namespace node. If there is no suitable element node in the result tree to accept a new namespace node, then an error occurs. Any use-attribute-sets attribute or template body is ignored.

TEAMFLY

If the current node is a text node, then the value of the current text node is copied to a new text node in the result tree. Any use-attribute-sets attribute or template body is ignored.

If the current node is a processing instruction node, then a processing instruction node is added to the result tree with the same target and data as in the current processing instruction node. Any use-attribute-sets attribute or any template body is ignored.

The <xsl:copy> element is useful when carrying out XML to XML transformations (see Chapter 6, Using XPath and XSLT to Produce XML).

The <xsl:copy-of> Element

The <xsl:copy-of> element is intended to copy a node set to the result tree. The <xsl:copy-of> element has one attribute, the select attribute, which is required and the value of which is an XPath expression. The <xsl:copy-of> element is always an empty element.

Thus if we wanted to copy to the result tree all <StockPrice> elements and their content, we could use <xsl:copy-of> like this:

```
<xsl:copy-of select="//StockPrice"/>
```

The major difference between the <xsl:copy-of> element and the <xsl:copy> element is that the <xsl:copy-of> element causes a deep copy (that is, the current node and all its descendant nodes are copied to the result tree).

One use of the <xsl:copy-of> element is when carrying out XML to XML transformations, but if one part of the source tree can be used unchanged in the result tree, then the <xsl:copy-of> element can be used to copy that sub-tree into the result tree unchanged.

The <xsl:element> Element

The purpose of the <xsl:element> element is to cause an element node to be created in the result tree. The <xsl:element> element can have three attributes. One, the name attribute, is required. The other two, the namespace attribute and the use-attribute-sets attribute, are optional. The value of the name attribute is an attribute value template that returns a QName.

Several techniques are available to add attributes to such a new element node. The use-attribute-sets attribute of the <xsl:element> element itself can be used. Alternatively, <xsl:attribute> elements, <xsl:copy> elements, or an <xsl:copy-of> element could be used.

A typical use of the <xsl:element> element would be in an XML to XML transformation to create an element in the output document for an attribute in the source document. Thus, if the source document looked like the following, we could apply the XSLT stylesheet in Listing 1.18.

```
<Book title="XPath Essentials" author="Andrew Watt" publisher="Wiley"/>
```

It is essentially the opposite transformation of that shown in Listing 1.5.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/">
<Book>
<xsl:for-each select="Book/@*">
<xsl:element name="{name()}">
<xsl:value-of select="."/>
</xsl:element>
</xsl:for-each>
</Book>
</xsl:template>

</xsl:stylesheet>
```

Listing 1.18 Transforming the Structure of Information about a Book (AttributesTo-Elements.xsl).

This will produce a restructured XML document with elements in the output replacing attributes in the source document:

```
<?xml version="1.0" encoding="utf-8" ?>
<Book>
<title>XPath Essentials</title>
<author>Andrew Watt</author>
<publisher>Wiley</publisher>
</Book>
```

Basically, what the template does is to take each attribute on the <Book> element in the source document and create a correspondingly named element in the output document, while inserting the value of the former attribute as the content of the newly created element.

The <xsl:fallback> Element

The purpose of the <xsl:fallback> element is to define what processing should occur if its parent element is not implemented by an XSLT processor. This might be used when an XSLT extension element was being used in an environment where it might not be supported in all XSLT processors.

The <xsl:fallback> element is an empty element. Its content is a template body.

Thus we might have code like this:

```
<ExtensionNamespace:NotSupportedHere>
<xsl:fallback>
```

```
<!-- What to do if the parent element is not supported. -->
</xsl:fallback>
<!-- What to do if the parent element IS supported. -->
</ExtensionNamespace:NotSupportedHere>
```

The <xsl:for-each> Element

The <xsl:for-each> element applies the same processing to each node in a set of nodes defined by an XPath expression in its select attribute.

The <xsl:for-each> element has one attribute, the select attribute, which is required. The content of an <xsl:for-each> element is a template body within which one or more <xsl:sort> elements may be optionally present and followed by the other part of the template body. The expression in the select attribute may not return the node set in the order desired for output; thus an <xsl:sort> element may be used as an immediate child of the <xsl:for-each> element. The <xsl:sort> element allows processing to be carried out in a specified order. It is permitted to have nested <xsl:sort> elements to allow sorting on more than one criterion.

For example, let's take a source document that describes a few of the books in the XML Essentials Series and process them so that they are copied to a new XML document but sorted in ascending order by the first name of the first author. To do that we use the <xsl:for-each> element with an appropriate <xsl:sort> element. First, Listing 1.19 is the source XML document that is ordered by the value of the title attribute.

Listing 1.20 is the XSLT stylesheet to process it. Note the <xsl:for-each> element and its nested <xsl:sort> element.

The stylesheet, when applied to the source document, produces the output (correctly ordered alphabetically by the first name of the first author) as shown in Listing 1.21.

The <xsl:if> Element

Sometimes you will want a template to be instantiated only if a certain condition applies. The <xsl:if> element allows such a test to be applied.

```
<?xml version='1.0'?>
<BookSeries>
<Book title="XHTML Essentials" author1="Michael Sauers" author2="Allen
Wyke" publisher="Wiley"/>
<Book title="XML Schema Essentials" author1="Allen Wyke" author2="Andrew
Watt" publisher="Wiley"/>
<Book title="XPath Essentials" author1="Andrew Watt" publisher="Wiley"/>
<Book title="XSL Essentials" author1="Michael Fitzgerald"
publisher="Wiley"/>
</BookSeries>
```

Listing 1.19 Some XML Essentials Books (BookSeries.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<BookSeries>
<xsl:for-each select="/BookSeries/Book">
<xsl:sort select="@author1" order="ascending" data-type="text"/>
<xsl:copy-of select="." />
</xsl:for-each>
</BookSeries>
</xsl:template>

</xsl:stylesheet>
```

Listing 1.20 An XSLT Stylesheet to Restructure BookSeries.xml (BookSeries.xsl).

```
<?xml version="1.0" encoding="utf-8" ?>
<BookSeries>
<Book title="XML Schema Essentials" author1="Allen Wyke" author2="Andrew
Watt" publisher="Wiley" />
<Book title="XPath Essentials" author1="Andrew Watt" publisher="Wiley"
/>
<Book title="XSL Essentials" author1="Michael Fitzgerald"
publisher="Wiley" />
<Book title="XHTML Essentials" author1="Michael Sauers" author2="Allen
Wyke" publisher="Wiley" />
</BookSeries>
```

Listing 1.21 The Output after Transforming (Sorting) BookSeries.xml. (BookSeriesSorted.xml)

The <xsl:if> element takes a single attribute, the test attribute, which is required. It is used like this:

```
<xsl:if test="SomeTest">
<!-- Instantiated if the test attribute returns true. Bypassed
otherwise. -->
</xsl:if>
```

The <xsl:if> element returns a boolean value of "true" or "false."

In practice, the <xsl:if> element could be used like this. In the example shown in Listing 1.22, we have a very simple set of data in an XML source document and we want only nonconfidential information to be output in an HTML page.

```
<?xml version='1.0'?>
<DataRepository>
<Information class="Public">This is public information.</Information>
<Information class="Public">Some more public information.</Information>
<Information class="Confidential">The public shouldn't see
this.</Information>
<Information class="Confidential">The public shouldn't see this
either.</Information>
</DataRepository>
```

Listing 1.22 Using the <xsl:if> Element (DataRepository.xml).

We can use the <xsl:if> element to test whether or not the class attribute indicates that it is information that should be available to the public, as in the stylesheet shown in Listing 1.23.

When the test in the <xsl:if> element is evaluated as the template is instantiated for each <Information> element, the content of the <Information> element is output if the class attribute has a value of "Public". Otherwise, the content of the <Information> element is ignored. As you can see in Figure 1.4, only the material suitable for public consumption is output.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Publicly available information.</title>
</head>
<body>
<h3>Publicly available information:</h3>
<xsl:apply-templates select="/DataRepository/Information"/>
</body>
</html>
</xsl:template>

<xsl:template match="Information">
<xsl:if test="@class='Public'">
<p><xsl:value-of select="."/></p>

</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Listing 1.23 The Stylesheet to Transform DataRepository.xml (DataRepository.xsl).

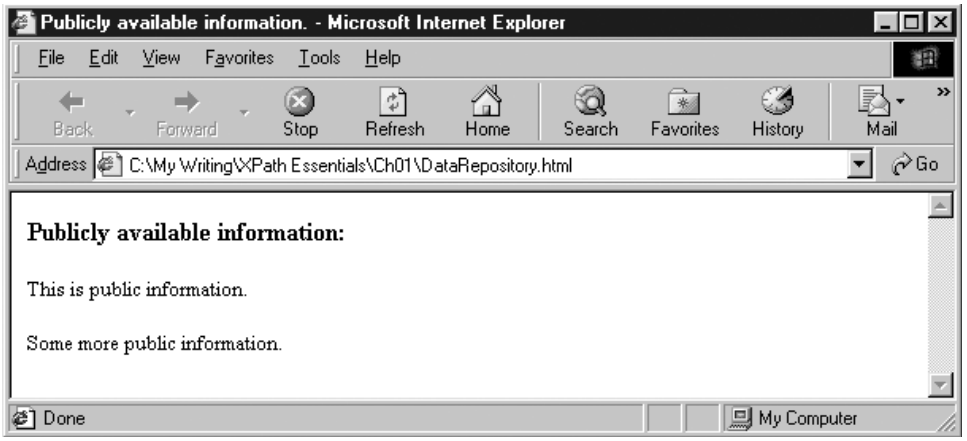

Publicly available information. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch01\DataRepository.html
Go
Publicly available information:
This is public information.
Some more public information.
Done
My Computer

Figure 1.4 The Output after Transforming DataRepository.xml to HTML.

The <xsl:number> Element

The <xsl:number> element has two purposes. It can be used to format a number for output and/or it can be used to output a sequential number to the current node (similar to an autonumber facility of a relational database management system).

The <xsl:number> element may take up to nine optional attributes:

- The *level attribute* determines the way in which a sequence number is allocated.
- The *count attribute* assesses which nodes are to be counted in order to arrive at a sequence number.
- The *from attribute* determines the start number in a sequence of numbers.
- The *value attribute* is a user-defined number for formatting.
- The *format attribute* defines the output format of a number.
- The *lang attribute* contains a language code, of the type defined in the XML 1.0 Recommendation for the xml:lang attribute.
- The *letter-value attribute* offers two options for numbering.
- The *grouping-separator attribute* defines which character is to be used to separate groups of digits (such as the use, in English, of the comma to separate millions from thousands, etc.). The grouping-separator varies between languages. In many European languages the period is the separator for thousands, rather than the comma as it is in English.
- The *grouping-size attribute* determines how many digits are in a group between each grouping separator. In English typically the grouping size is three.

The <xsl:otherwise> Element

The <xsl:otherwise> element is used within an <xsl:choose> element to indicate the processing that is to take place in the absence of a test being satisfied in any <xsl:when> element nested within the <xsl:choose> element.

The <xsl:otherwise> element has no attributes and its content is a template body.

The <xsl:processing-instruction> Element

The purpose of the <xsl:processing-instruction> element is to output a processing instruction node to the result tree.

An <xsl:processing-instruction> element has one required attribute—the name attribute—which defines the target for the processing instruction. The content of an <xsl:processing-instruction> element typically constitutes the data of the processing instruction.

A widely used processing instruction is the <?xml-stylesheet?> processing instruction which is used in an XML source document to associate a stylesheet with that source document. Thus if we wanted to output a processing instruction that associated an XSLT stylesheet with the output document, we could do so using code like this:

```
<xsl:processing-instruction name="xsl-stylesheet">
<xsl:text>href="ALinkedXSLTStylesheet.xsl" type="text/xsl"</xsl:text>
</xsl:processing-instruction>
```

Alternatively, if it was desired to associate a Cascading Style Sheets stylesheet with an XML document, then the following could be used:

```
<xsl:processing-instruction name="xsl-stylesheet">
<xsl:text>href="ACascadingSheet.css" type="text/css"</xsl:text>
</xsl:processing-instruction>
```

The <xsl:sort> Element

The purpose of the <xsl:sort> element is to provide sorting functionality in XSLT stylesheets. The <xsl:sort> element is used within <xsl:apply-templates> and <xsl:for-each> elements.

The <xsl:sort> element may have five attributes, none of which are required. The select attribute indicates the sort key to be used, which, by default, is the string value of the context node. The order attribute defines the order in which nodes are to be processed and may take the values “ascending” or “descending.” The case-order attribute determines how the case of characters is to be handled in sorting, and it may take the values of “upper-first” or “lower-first.” The lang attribute is a two-character language code. The lang attribute is important since alphabetical ordering of characters is not the same in all languages which use the Roman alphabet, for example. The data-type attribute defines whether the values to be sorted are to be treated as text or as numbers. The data-type attribute may take the values of “text” or “number” or it may use a user-defined data type indicated by a QName.

The <xsl:text> Element

The purpose of the <xsl:text> element is to output literal text to the result tree. It has one attribute, the disable-output-escaping attribute, which can take the values of "yes" and "no."

If the disable-output-escaping attribute has the value of "yes," then the content of the <xsl:text> element should be output without any escaping of special characters.

Any whitespace contained within an <xsl:text> element is output as is. In the absence of the <xsl:text> element, such whitespace nodes are output only if an ancestor element has the xml:space attribute with a value of "preserve" with no intervening element having a contradictory value for that attribute.

The <xsl:value-of> Element

The purpose of the <xsl:value-of> element is to write the string value of an expression contained in its select attribute to the result tree.

The <xsl:value-of> element has a required select attribute and an optional disable-output-escaping attribute.

The <xsl:value-of> element has been used in earlier examples and will be used in many examples later in this book so I won't develop further examples here.

The <xsl:when> Element

The purpose of the <xsl:when> element is to provide a means to test for a particular situation within an <xsl:choose> element.

The <xsl:when> element has a required test attribute that evaluates to a boolean value. The content of an <xsl:when> element is a template body that is instantiated if the test attribute of the <xsl:when> element evaluates to "true."

The <xsl:with-param> Element

The purpose of the <xsl:with-param> element is to set the value of parameters when using the <xsl:apply-templates> element or when calling a template using the <xsl:call-template> element.

The <xsl:with-param> element has a required name attribute that reflects the name of the parameter. It may also have an optional select attribute, which evaluates to the parameter supplied. If the <xsl:with-param> element has a select attribute, then it should be an empty element. If the <xsl:with-param> element has no select attribute, then its content should provide the value of the parameter.

Attribute Value Templates

An attribute value template provides a means to determine at run time the value of an attribute.

The syntax for an attribute value template is an XPath expression contained within curly braces. Thus, the syntax would look like this

```
MyAttribute="{$SomeVariable}"
```

or like this

```
MyAttribute=""{/Book/Chapter[position( )=1]/Paragraph[last( )/Footnote}"
```

On literal result elements the attribute value template provides a more succinct way to define the value of an attribute. An alternative, but more verbose, way to achieve the same thing would be to use the <xsl:attribute> element described earlier.

On a limited number of XSLT elements some attributes can be expressed as an attribute value template, thus allowing the value of the attribute to be determined at run time.

In either case, the attribute value template is instantiated by replacing the attribute value template and its surrounding curly braces with the string value of evaluation of the expression contained within the curly braces.

For a more detailed description of attribute value templates and how to use them, see Chapter 7.6.2 of the XSLT 1.0 Recommendation.

XSLT Tools

There are a number of XSLT processors available, frequently at no cost. In this section I will briefly describe some of the available tools. If you are already using an alternative XSLT processor and are happy with it, feel free to use it as you try out the examples later in the book.

The two command line XSLT processors I will describe are the Saxon XSLT processor created by Michael Kay and the Xalan XSLT processor from the Apache Organization. Both of these are Java-based programs. In addition, you may want to explore the Microsoft MSXML3 combined XML parser and XSLT processor.

Saxon and Instant Saxon

The Saxon XSLT processor is an open-source product created by Michael Kay, currently a member of the W3C's XSL Working Group. It is available for download from http://users.iclway.co.uk/mhkay/saxon, in several versions. Look for a version that is marked stable and avoid versions marked as "latest"—since they will most likely contain bugs—unless, of course, you want to explore the cutting edge of XSLT.

As well as being available in several numerical versions, Saxon is also available in two major forms. Saxon is a cross-platform Java version, which includes source code, documentation, and sample files. Instant Saxon is a Windows-executable version without source code or sample files but which can be installed rapidly on 32-bit Windows operating systems.

The full version of Saxon requires a Java Development Kit or run time environment, which can be downloaded from http://java.sun.com. It is recommended that you use Java 1.3, rather than earlier versions, although Saxon will run with JDK 1.1 or 1.2. Saxon appears to run about three times as fast under the Sun JDK 1.3 as under the Windows JVM; therefore, particularly for larger files, you will probably want to ensure that you have the Sun JDK installed.

Instant Saxon uses the Java Virtual Machine included in Windows. Saxon's author says that it performs significantly better using a full JDK or JRE than Instant Saxon does using the Windows JVM. For the purposes of this book, Instant Saxon is more than adequate but for production use, you are likely to be better served by using the full version of Saxon.

To install the full version of Saxon, choose the latest stable version from the Web site, then download to a suitable location on your machine. Unzip the file using WinZip, or similar program, to a suitable directory, such as c:\FullSaxon.

Since I have lots of Java versions and programs, I tend to create small batch files set the CLASSPATH to something very simple, to minimize the risk of conflicts of versions of various files littering my development machines. In this instance, I created a simple batch file, called FullSaxon.bat, so that if I open an MSDOS window I can run fullsaxon.bat and know that the CLASSPATH is appropriate and simple.

```
SET CLASSPATH=.;c:\fullsaxon\saxon.jar
```

If you want to test that it has run correctly, simply type:

```
ECHO %CLASSPATH%
```

at the command line and you should see that the CLASSPATH has been set as just described.

Having set the CLASSPATH (and, of course, ensuring that your Java Virtual Machine is in the PATH), move to the c:\fullsaxon directory and issue the following command at the command line

```
java com.icl.saxon.StyleSheet
```

You should see a series of messages from Saxon similar to those in Figure 1.5. The messages tell you that you don't have your syntax complete but the fact that the messages are coming from Saxon indicates that it is behaving correctly in response to what you typed. The first line, "No source file name" indicates what you did wrong. The following lines offer hints about a more complete or correct syntax to use.

NOTE **If you receive an error message like "Exception in thread 'main' java.lang.noClassDefFoundError: com/icl/saxon/Stylesheet", then it is likely you have mistyped the command just mentioned above. Be particularly careful to notice "StyleSheet" with two uppercase S characters. Remember that Java is case sensitive.**

If we want the full version of Saxon to use the Invoice.xml file from Chapter 2 to apply the Invoice.xsl stylesheet and produce a file Invoice.html, we would issue the following command:

```
java com.icl.saxon.StyleSheet Invoice.xml Invoice.xsl > Invoice.html
```

TEAMFLY

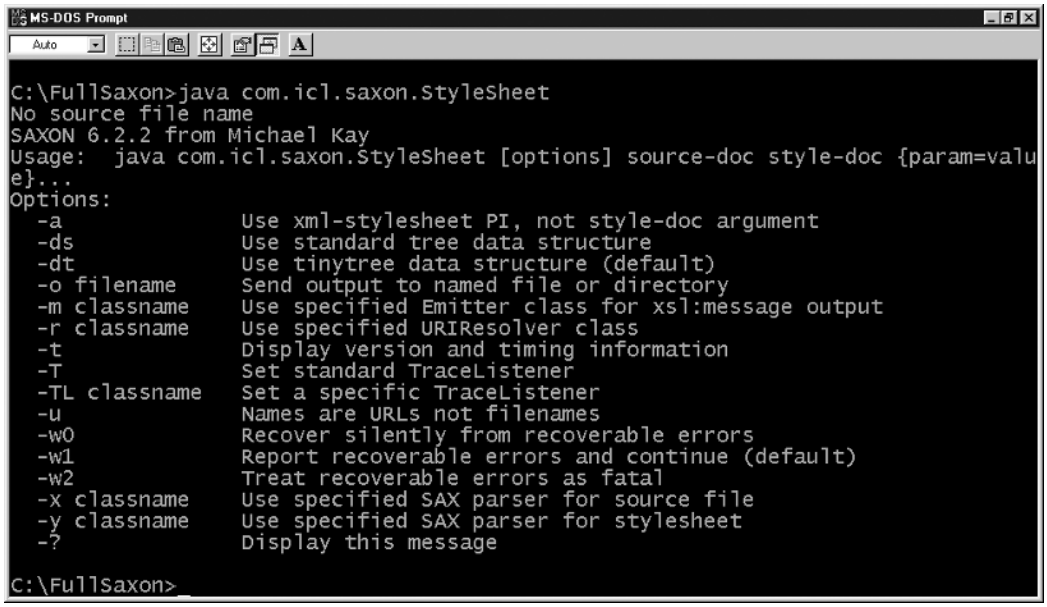

MS-DOS Prompt
Auto
C:\FullSaxon>java com.icl.saxon.StyleSheet
No source file name
SAXON 6.2.2 from Michael Kay
Usage: java com.icl.saxon.StyleSheet [options] source-doc style-doc {param=valu
e}...
Options:
-a Use xml-stylesheet PI, not style-doc argument
-ds Use standard tree data structure
-dt Use tinytree data structure (default)
-o filename Send output to named file or directory
-m classname Use specified Emitter class for xsl:message output
-r classname Use specified URIResolver class
-t Display version and timing information
-T Set standard TraceListener
-TL classname Set a specific TraceListener
-u Names are URLs not filenames
-w0 Recover silently from recoverable errors
-w1 Report recoverable errors and continue (default)
-w2 Treat recoverable errors as fatal
-x classname Use specified SAX parser for source file
-y classname Use specified SAX parser for stylesheet
-? Display this message
C:\FullSaxon>

Figure 1.5 The Help Messages Output by the Full Version of Saxon.

NOTE **If you download the full version of Saxon to a Windows platform, it may sometimes lack a .zip extension. To remedy this, right-click on the icon for the file, choose Rename, and add a ".zip" to the filename and save that back to disk. That seems to fix the problem, so that you can unzip the file with WinZip or similar program.**

To install Instant Saxon from the Web, simply right-click on the appropriate stable version and choose download to disk. A zip file will be downloaded to your machine. Unzip it with WinZip or similar program and install Instant Saxon to an appropriately named directory, such as c:\InstantSaxon.

To test whether you have installed Instant Saxon correctly, open an MSDOS window, and change directory to the c:\InstantSaxon directory (or your installation directory). Type Saxon on the command line. If the installation was correctly done, then you should see an error message similar to the one shown in Figure 1.5. The message tells you, in effect, that you need to issue commands more complex than just "Saxon."

If we wanted to use Instant Saxon to process Invoice.xml from Chapter 2, with Invoice.xsl as the stylesheet and to produce Invoice.html as the output document, we would issue the following command on the command line:

```
saxon Invoice.xml Invoice.xsl > Invoice.html
```

A command in that form will allow you to apply any named stylesheet to a source document, overriding any <? xml-stylesheet ?> processing instruction contained in the source XML document.

Xalan

Xalan is an XSLT processor available for free download from the Apache Organization, well-known for server and other software. Xalan can be used from the command line or you can alternatively use it as a servlet or applet.

To download the Xalan XSLT processor, go to http://xml.apache.org. In the navigation bar at the left of the Web page there are three links to Xalan: two for Java versions and one for a C++ version. Choose the link which is appropriate to your computer configuration or needs.

NOTE Be aware that the Xalan XSLT processor is an evolving tool, as it progressively incorporates more of the W3C Recommendations that impinge on XSLT and XPath. Read the introductory material on the Apache Web site carefully so that you are clear about which of the many versions available for download are stable versions and which are more experimental.

To Run Xalan 1.2.2 you will need a Java Development Kit or a Java Runtime Environment version 1.1.8 or higher. If you don't already have one, then it can be downloaded from http://java.sun.com. Make sure you add the location of the Java compiler to the PATH statement in your autoexec.bat file.

You will need to add the directory in which you install Xalan.jar (and its accompanying Xerces.jar file) to the CLASSPATH on your computer. Or, alternatively, create a small batch file to use when running Xalan and Xerces.

NOTE If you are using JDK 1.1.8, then be sure to also add the classes.zip file to the CLASSPATH.

Alternatively, you can use the newer Xalan-Java version 2. There are also links to it from the xml.apache.org home page.

To use Xalan from the command line you use a command like this

```
java org.apache.xalan.xslt.Process -in xmlSource
   -xsl stylesheet -out outputfile
```

Of course, you substitute appropriate filenames in the parts that are italicized above. So, if we were to try to process the Invoice.xml file in Chapter 2 with the Invoice.xsl file as the stylesheet, and wanted to create the output file Invoice.html, we would issue the following command. (Make sure the Java virtual machine has already been added to the PATH statement and the CLASSPATH has been set to include the directory in which the Xalan.jar and Xerces.jar files had been located.)

```
java org.apache.xalan.xslt.Process -in Invoice.xml -xsl Invoice.xsl -out
   Invoice.html
```

I have lots of different Java settings on my development machines and therefore have various batch files to create appropriate CLASSPATHs for using different Java applica-

tions. If I wanted to ensure that only the versions of Xerces and Xalan located in, say, the Xalan-J 2.1.0 directory were in the CLASSPATH, then I would have a short batch file like the following, assuming that I had installed Xalan in a subdirectory of c:\xalan.

```
SET CLASSPATH=.;c:\xalan\xalan-j_2_1_0\bin\xalan.jar; c:\xalan\xalan-
   j_2_1_0\bin\xerces.jar;
```

After running the batch file, then I know that the only Java files that will be accessed are in the current directory and in the bin directory where xalan.jar and xerces.jar are located.This will avoid possible conflicts of applications, or versions, when you have an autoexec.bat file with a series of statements which add more and more to the CLASSPATH.

Microsoft MSXML3

The Microsoft MSXML product shipped with the Internet Explorer browser incorporates an XPath processor and an XSL processor as well as an XML parser. However, there is one important caution here: all versions of MSXML prior to MSXML3 used a now obsolete Microsoft dialect of an early draft of XSLT. That outdated dialect was shipped with Internet Explorer 5.0 and 5.5.

NOTE Internet Explorer versions 4, 5, and 5.5 all shipped with what are now obsolete versions of MSXML. If you want to properly learn and use XPath you need to upgrade to MSXML3. To recognize stylesheets that use the outdated version, look for a namespace declaration like this:

xmlns:xsl="http://www.w3.org/TR/WD-xsl"

while stylesheets which use the correct W3C versions of XPath and XSLT will use

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

If you want to use the W3C official versions of XPath and XSLT, then you will need to upgrade to at least MSXML3. You can download and install MSXML3 without changing your version of Internet Explorer.

MSXML3 provides an easy way to view an XML source document already processed by its associated stylesheet. If you use the <?xml-stylesheet?> processing instruction within the XML source document with an XML to XML transformation, then the output XML is directly viewable in the Internet Explorer browser when opening the source XML document.

NOTE There is a very useful "unofficial" support site for MSXML located at www.netcrucible.com/, which has lots of useful information on MSXML3, including installation issues, and so on. In reality, it seems to be run by a Microsoft employee.

The NetCrucible Web site provides lots of useful information about how to install MSXML3 and how to run it in "side-by-side" mode or "replace" mode. See the site for further information.

NOTE **A Preview Release of MSXML4 is available for download from the Microsoft Web site. Visit http://msdn.microsoft.com/downloads/default.asp?URL=/code/sample.asp?url=/MSDN-FILES/027/001/594/msdncompositedoc.xml for further information. Unless you are familiar with the issues that may arise from using Beta software, I suggest you at least initially ensure that you use MSXML3.**

Oracle XML Development Kit

Another XSLT processor is available for download from Oracle. The XML Development Kit is described at http://technet.oracle.com/tech/xml/, and there are numerous links from that page to Oracle's range of XML products.

Looking Ahead

Having spent time reviewing the fundamentals of XML and of XSLT, both of which are highly relevant to XPath and how to use it, let's move on to take a closer look at XPath itself in Chapter 2.

XPath Fundamentals

If you have many megabytes of data stored as XML, you want to be able to select which parts of that data you send to a business partner for display on a Web site or to use for some other purpose. XPath and its expressions and location paths are designed to help you make appropriate selections from your XML data.

This chapter is designed to introduce you to the fundamentals of the XML Path Language without going into great depth on any particular point. Its purpose is to communicate the framework of XPath so that you can hang detail on it as you work through the later chapters. Many of the points mentioned in this chapter will be developed in greater depth in later chapters, often with plenty of illustrative example code.

I have found XPath to be a technology you need to see as a whole to be able to fully understand the parts. When I was first learning XPath, that was what I found most difficult, and I know others do too. If you are new to XPath and are finding it difficult to see exactly where and how the parts of XPath fit together, please work your way carefully through the chapter. If you find that you just can't "see" some aspect of XPath as you read about it, move on and you may see how it fits when you look at it from another angle later in the chapter. As you see things from various angles, the whole picture should progressively become clearer in your mind. If you find parts of XPath pretty dry and abstract, then feel free to skip ahead. As you get a fuller impression of all that XPath is about, you can come back and take another look at these dry parts and hopefully see their relevance more clearly.

XPath Overview

In this section we will take a very general view of what XPath is and what it does. In later parts of the chapter I will go on to introduce aspects of XPath in more detail as a preparation for really detailed examination of XPath in later chapters.

The XML Path Language (XPath) is designed to help you—and an XSLT processor—find your way around XML documents, so that you can select a defined part of an XML document for any of a number of purposes. Examples might be for display as XML using Cascading Style Sheets, conversion of selected parts of the XML to HTML for display, transformation into another type of XML document in a B2B data interchange scenario, or to display the result of a query of the XML document.

XPath is of crucial importance to all XSLT transformations. If you can't define and locate a particular element, or, more specifically, the node which represents it, then you can't process it in the correct way to produce the desired output. Similarly, the XPath function library provides support for some numeric, string, and Boolean functions as well as the node-set functions that are particularly relevant to XPath's use as a navigation tool. Those functions are designed to help you select which nodes are to be processed in a particular way.

NOTE If you are aware of gaps in your knowledge of the fundamentals of XML and XSLT and you skipped Chapter 1, please be sure to look at it so that you are up to speed on at least the basics of both technologies, since XPath depends on XML rules and is regularly used with XSLT.

One of the fundamental things to realize about XPath is that it is not used alone. It is used in conjunction with other XML-based technologies. At the present time the dominant use of XPath is with the Extensible Stylesheet Language Transformations (XSLT).

XPath was also designed to be used with the XML Pointer Language, XPointer, which at the time of writing remains at Working Draft status at the World Wide Web Consortium.

NOTE If you want to take a look at the XSLT and XPointer specifications, they are located on the W3C Web site. XSLT 1.0 is a full Recommendation located at www.w3.org/TR/1999/REC-xslt-19991116. XPointer is currently a "last call" Working Draft located at www.w3.org/TR/2001/WD-xptr-20010108. Any update to the XPointer specification since the time of writing will be located at www.w3.org/TR/xptr. If that URL displays the Working Draft of January 8, 2001, then no new version has been issued.

A second fundamental point is that XPath, while used solely with XML application languages, is not written using XML syntax. For example, the following XPath expression is clearly not written in XML.

```
child::chapter/section[position()=1]/paragraph[position()=2]
```

You will perhaps notice in that XPath expression a similarity to the way a path, as in a directory listing or in a URL, is written. That similarity to other path notations gives rise to XPath's name. It is a path language *for* XML, not a path language *in* XML.

Using a non-XML syntax makes it easier to use XPath in Uniform Resource Identifiers (URIs) and within the values of attributes on XML elements. The permitted characters in URIs and in XML are not identical and using a non-XML syntax for XPath simplifies using XPath in URIs. Similarly, if XPath were expressed in XML syntax and an XPath expression were present within an XML attribute, it would be necessary to escape some of the characters, adding to the complexity and cumbersomeness of creating XPath expressions.

The XPath Forms of Syntax

XPath has, fairly confusingly, four forms of syntax, which may be used to write expressions and location paths.

At this stage I will simply mention the meaning of many of the location paths without a detailed analysis of why they mean what I say they mean. I will cover each of the four forms of XPath syntax in some detail in Chapter 4, "The Four XPath Syntaxes."

NOTE **A location path is an XPath expression that selects a node-set.**

XPath can be viewed as a way to navigate round XML documents. Thus XPath has similarities to a set of street directions. When you are receiving street directions, you need to know what your starting point is; otherwise, instructions like "Go east 3 blocks, turn right, and look for the second gray building on the left" don't make much sense. In XPath the starting point is called the *context node*. The logical parts of the in-memory representation of an XML document are termed *nodes*. An XPath processor deals with nodes, not with the more familiar elements and attributes.

In a set of street directions, you will often be told to go first in a particular direction, "Go east 3 blocks ...," for example. In XPath the equivalent of a direction is called an *axis*. XPath has a total of 13 different axes, which we will look at in more detail later. A particularly commonly used axis is the child axis. Thus for the simple XML document in Listing 2.1, if the node representing the <Invoice> element were the context node, then the XPath expression

```
child::*
```

which selects all child element nodes of the context node would select the nodes that represent the <CustomerName>, <Address>, and <City> elements. If we wanted to select only the <CustomerName> element, we could use a more specific XPath expression

```
child::CustomerName
```

which selects only child elements of the context node that have an element type name of CustomerName.

```
<?xml version='1.0'?>
<Invoice>
 <CustomerName>
   John Smith
 </CustomerName>
 <Address>
   123 Any Street
 </Address>
 <City>
   Anytown
 </City>
</Invoice>
```

Listing 2.1 A Simple Invoice in XML (Invoice.xml).

XPath can also select a <CustomerName> element from our example XML document using yet another syntax

```
/child::Invoice/child::CustomerName
```

which can be abbreviated to

```
/Invoice/CustomerName
```

Already you have seen three out of the four forms of syntax that can be used in XPath. The fact that XPath has four syntax variants is one thing that initially makes it difficult to understand. Let's use the one you have just seen in a simple practical example.

If, for example, we wanted to use XSLT to transform our source XML document to a simple HTML document that displays the customer name, then we could use a simple XSLT stylesheet like that in Listing 2.2 to produce a very simple HTML document shown in Listing 2.3.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using XPath to display a customer name</title>
</head>
<body>
<br />
```

Listing 2.2 An XSLT Stylesheet to Transform Invoice.xml (Invoice.xsl).

```
The customer name is: <xsl:value-of select="/Invoice/CustomerName"/>
<br />
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.2 (*Continued*)

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="application/xml;
  charset=utf-8">
      <title>Using XPath to display a customer name</title>
   </head>
   <body><br>
      The customer name is: John Smith
      <br></body>
</html>
```

Listing 2.3 The HTML Output Showing the Result of Transforming Invoice.xml (Invoice.html).

If you are using the Instant Saxon XSLT processor to try out this example and if you have the source file, Invoice.xml; the stylesheet, Invoice.xsl; and the Saxon processor in the same directory, then the command

```
Saxon Invoice.xml Invoice.xsl > Invoice.html
```

will create a very simple HTML document, Invoice.html, the source code for which is shown in Listing 2.3. If you have chosen to type in the code and have done so correctly and have correctly expressed the command line correctly, then a simple HTML file should be produced.

The line

```
<xsl:template match="/">
```

contains the simplest of all XPath expressions, the character "/" matches the root node of the tree which exists in memory as the representation of a source XML document.

From the point of view of XPath, the following line in the XSLT stylesheet is what does the work:

```
The customer name is: <xsl:value-of select="/Invoice/CustomerName"/>
```

with the XSLT <xsl:value-of> element indicating that the text content of the node representing the CustomerName element is to be displayed. The select attribute of the <xsl:value-of> element uses the XPath expression

```
/Invoice/CustomerName
```

to indicate that it is the value of the <CustomerName> element which is to be displayed.

To be able to understand what is happening when an XPath expression is being evaluated, we need to take a closer look at the in-memory hierarchical tree which represents the source XML document and at the nodes which make up that hierarchy.

The XPath Data Model

Earlier in this overview I casually referred to a "node" being processed without defining the term node.

If you read the XML 1.0 specification, you will find absolutely no mention of nodes, except for a single mention in Appendix E. If nodes are so important for XPath, and if XPath is used only with XML, and the XML Recommendation doesn't really tell us anything about nodes, what is going on?

To draw out what's going on, let's take a close look at the simple XML document in Listing 2.4.

This document consists of one character after another—a series of characters. That is why it can be called a *serialized file* or document; it is, at bottom, only a series of characters.

But when we read that file we don't treat it as a "<" character followed by a "?" character, followed by an "x" character and so on. We interpret it as a series of *logical tokens*. For example, in the first line of the code there are no prizes for being able to recognize the whole line as an XML declaration. Similarly we take it almost for granted that the second line is the start tag of an element whose end tag is in the last line of the code.

What we are doing is translating a series of characters into a logical representation of that sequence of characters, because we can see that the sequence of characters isn't merely in some random order but actually conforms to XML 1.0 syntax. For XPath (and XSLT) to be able to process an XML document, its logical structure needs to be presented to the processor, but not as elements, attributes, an XML declaration, etc., but in

```
<?xml version='1.0'?>
<Book>
<Introduction>Hello and welcome.</Introduction>
<Chapters>
<Chapter number="1">Some content.</Chapter>
<Chapter number="2">Some more content on another topic.</Chapter>
</Chapters>
</Book>
```

Listing 2.4 A Simple Description of a Book in XML (Book.xml).

a hierarchical logical representation in memory. This is often called a "tree," although technically it need not conform exactly to a tree, as the term is used strictly in computing science.

Our source document, when schematized on paper to represent what the in-memory tree is like, would look similar to what you see in Figure 2.1. I have drawn each node as an oval, although there is no significance at all in the shape chosen—nodes don't have a shape.

Starting from the top of the diagram, the first thing we see is a representation of the *root node*. If you look at the original XML document you will see no direct sign of a root node. But as we think of the document we implicitly place the XML declaration and the elements that follow inside a container which we casually may call a document or a file

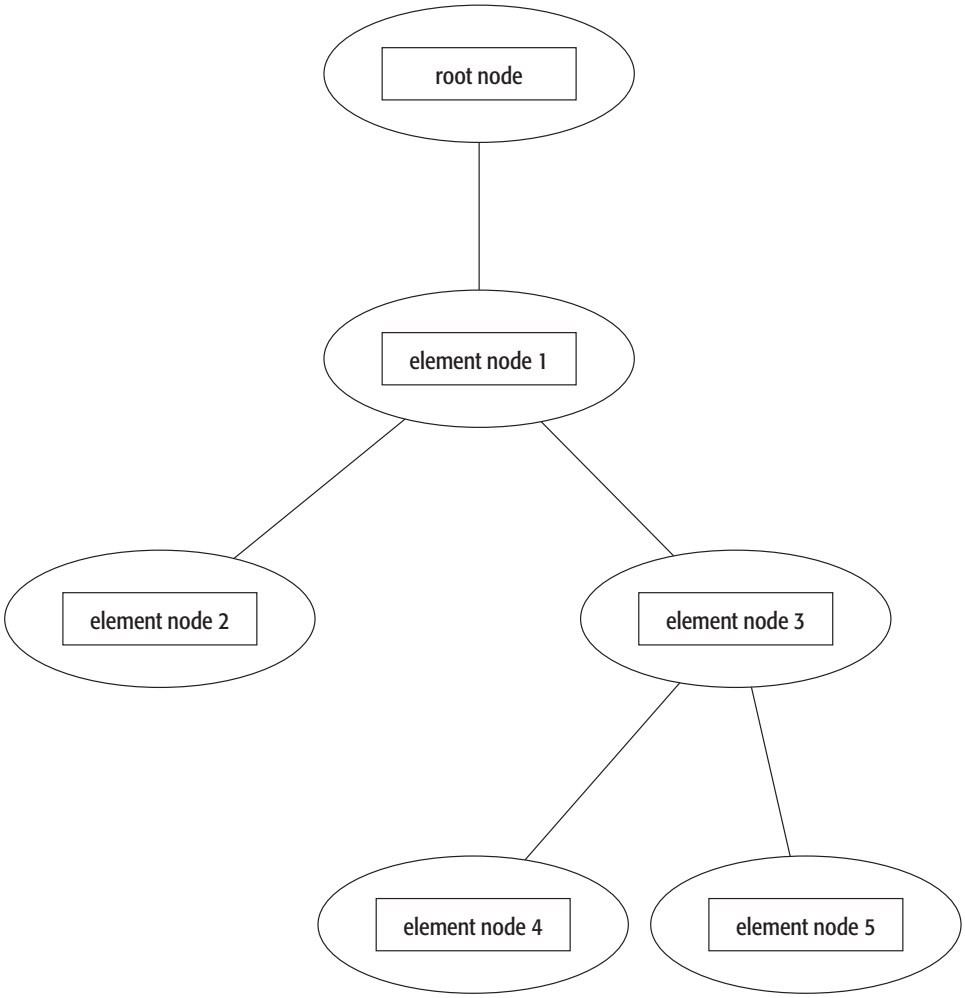


Figure 2.1 A Schematic of the Source File as a Node Hierarchy.

but which in XML jargon is a document entity. The root node is, for most practical purposes, the XPath representation of the document entity, at least in the sense that it is a container for all other parts of the document or its representation.

The line from the root node to the node I have labeled "element node 1" represents the relationship between the root node and its child element node, which represents the <Book> element in our XML document.

If you look back at the XML document you will see that the <Book> element has nested within it an <Introduction> element and a <Chapters> element. In the serialized document we refer to it as *nesting of elements*. In the XPath data model we say that the element node representing the <Book> element has two element node children—one of which represents the <Introduction> element (element node 2 in the diagram) and another node which represents the <Chapters> element (element node 3 in the diagram).

In the source document you can see that nested within the <Chapters> element there are two <Chapter> elements. Thus, in the diagram, element node 3 (which represents the <Chapters> element) has two element node children—element node 4 (which represents the first <Chapter> element) and element node 5 (which represents the second <Chapter> element).

So, having taken a brief look at a very simple hierarchy of nodes, let's take a look at the "workflow" of what happens when XPath is applied in an XSLT transformation (see Figure 2.2).

The left-hand box represents the XML source document in its serialized form, which you saw a short time ago as the file Book.xml (Listing 2.4). When an XML document is parsed, it is converted in memory into a hierarchy of nodes, as in the second box which shows the *source tree*.

At this stage nothing has been transformed (at least not in the XSLT sense). If that tree of nodes is changed back into a serialized form, it will be identical to (or very similar to) the XML document which we started with. Any differences will be trivial and will not affect the meaning of the document. For example, we might have used single quotes on attribute values and it might have double quotes when written out again. However, our intention is not to write the XML document out again unchanged but to carry out an XSLT transformation.

The XSLT transformation is represented in the diagram by the arrow from the box representing the source tree to the box representing the result tree.

The final step shown in the diagram is the serialization of the result tree into an *output document*, also referred to as a *result document*.

However, you should be aware that serialization to a result document need not happen after a single transformation. The result tree can be the input into another XSLT transformation. However, for purposes of simplicity, all the examples in this book will follow the pattern of the four stages: the serialized XML document is converted to a

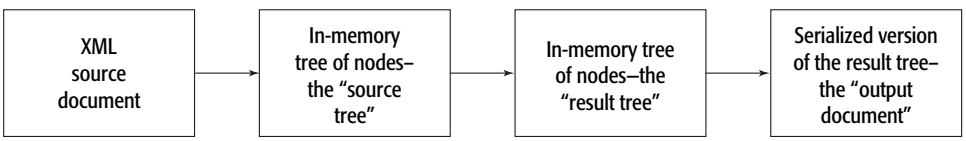

XML source document
In-memory tree of nodes– the "source tree"
In-memory tree of nodes–the "result tree"
Serialized version of the result tree– the "output document"

Figure 2.2 Schematic of Workflow during an XSLT Transformation.

hierarchical tree in memory, and the "source tree" is transformed (in the XSLT sense) into another hierarchical structure (the "result tree"), which is then serialized into an output document.

NOTE **I will use the term *result tree* synonymously with *output tree*. Similarly *result document* will be treated as meaning the same as the term *output document*.**

Having taken a general look at the notion of what a node is in XPath and where the trees of nodes fit into an XSLT transformation, let's take a closer look at some aspects of exactly what XPath nodes are available to us within the XPath in-memory hierarchy and what their characteristics are.

XPath Nodes

In this section I will describe the XPath node types. As mentioned earlier, a source XML document is converted in memory into a hierarchical tree of nodes. Details of how nodes are constructed are also discussed in Chapter 3, "XPath Data Model," which describes more fully the XPath data model.

An XPath document has potentially seven types of nodes:

1. Root node
2. Element nodes
3. Attribute nodes
4. Namespace nodes
5. Processing-instruction nodes
6. Comment nodes
7. Text nodes

Each tree, in XPath terminology, has a *root node* as the source of the tree from which a branching structure hangs. A root node may not occur anywhere else other than at the root of the tree. Thus each tree may have only one root node.

An *element node* represents an element in the serialized source XML document. An element node may have an ordered list of child element nodes. The fact that the child element nodes are ordered is useful when we will later want to select a node (or a node set) according to the position of the nodes within that ordering.

So, if we have a source document like this

```
<Node1>
  <Node2>Some content</Node2>
</Node1>
```

wherein the serialized version of the XML document the <Node2> element is nested within the <Node1> element, in the in-memory XPath hierarchy the element node

representing the <Node2> element is a child element node of the element node representing the <Node1> element.

An *attribute node* represents an attribute in the source XML document. An attribute node has a parent element node. However, in XPath terminology an attribute node is not, contrary to what you might have expected, a child of its parent element node. The reason this arises is because of the way that the XPath axes were established when the specification was written. The child axis in XPath is defined as not containing either attribute or namespace nodes. So attribute nodes are not children of their parent node.

A *namespace node* represents a namespace that is in scope on the element in the source document represented by parent element node of the namespace node. An element node will have a namespace node for each namespace URI that is in scope on the element.

A *processing-instruction node* represents a processing instruction in the source XML document.

A *comment node* represents in the in-memory hierarchy a comment in the source XML document.

A *text node* represents the text content of an element, as here

```
<AnElement>Here is some text content.</AnElement>
```

or may represent the content of a CDATA section, as here

```
<![CDATA[Here is some text contained in a CDATA section.]]>
```

I mentioned earlier that there could be minor changes if a source tree were simply changed back into a serialized document. Here is another instance of the changes that can occur. A text node could have arisen from either of the pieces of syntax shown. There is no way to distinguish the text nodes that would be created in memory. Thus, if the CDATA section contained tags, or other content that requires to be escaped, and was written back as element content, a parsing error could easily occur.

NOTE The W3C has recognized the issues just mentioned and has developed a specification for "Canonical XML" which provides a form of a document independent of minor syntax options. The Recommendation is located at www.w3.org/TR/xml-c14n. Canonicalization is sometimes, in the jargon, referred to as "c14n." The use of XPath in Canonical XML is described in Chapter 11, "XPath in Canonical XML and XML Signatures."

Having looked briefly at the types of node available in XPath let's look at how XPath allows us to choose such nodes and perform other processing.

XPath Expressions

An XPath expression is the most general type of XPath statement. An XPath expression can yield one of the following four types of objects:

Node set. An ordered set of nodes

String. A sequence of characters

Number. A floating point number

Boolean. "true" or "false"

When the object yielded by an XPath statement is a node set object, then the XPath expression is said to be an XPath location path. The XPath location path is, in practice, the most important type of XPath expression, and we will look in more detail at location paths later in the chapter.

Context Node

Evaluation of an XPath expression takes place in a context. You can think of that as being the place where you presently are. It is similar to being on a street. What you see across the street depends on where you are. The context position within the node hierarchy is exactly that. It simply tells you where you are. Once you know where you are, you are in a position to think about how to get to where you want to go.

Just as describing your position on the street can be quite lengthy, "I am near 5th Avenue and 8th Street just along from the Post Office opposite the department store," so an XPath context can be quite lengthy. A context in XPath has the following parts:

- A node (termed the *context node*)
- A context position (a non-zero positive integer)
- A context size (a non-zero positive integer)
- A set of variable bindings
- A function library
- The set of namespace declarations that are in scope for the context node

That quite possibly sounds pretty opaque, so let's break that list down while we look at some examples. If we have a simple XML document like that shown in Listing 2.5, which we want to transform for presentation as HTML, we can use an XSLT stylesheet as shown in Listing 2.6.

The output HTML will look like that shown in Listing 2.7.

```
<?xml version='1.0'?>
<MediaCompanies>
<Company abbreviation="AOL">America OnLine</Company>
<Company abbreviation="CNN">Cable News Network</Company>
</MediaCompanies>
```

Listing 2.5 An Inventory of Media Companies (MediaCompanies.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>Media Companies</title>
</head>
<body>
Here are some important media companies.
<xsl:apply-templates select="MediaCcompanies/Company"/>
</body>
</html>
</xsl:template>

<xsl:template match="Company">
<p><b><xsl:value-of select="."/></b> whose abbreviated name is
  <xsl:value-of select="@abbreviation"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.6 An XSLT Stylesheet to Transform MediaCompanies.xml to HTML (MediaCompanies.xsl).

Let's take a closer look at what happened to the context node during the XSLT transformation.

The first step is that the XSLT processor looks for an <xsl:template> element that has a match attribute equal to the root node, like this

```
<xsl:template match="/">
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Media Companies</title>
</head>
<body>Here are some important media companies.
<p><b>America OnLine</b> whose abbreviated name is AOL</p>
<p><b>Cable News Network</b> whose abbreviated name is CNN</p>
</body>
</html>
```

Listing 2.7 The HTML Produced Following the Transformation of MediaCompanies.xml (MediaCompanies.html).

Thus the initial context node is the root node of the source tree.

Then the XSLT processor encounters several lines of code, which are to be output literally.

```
<html>
<head>
<title>Media Companies</title>
</head>
<body>
Here are some important media companies.
```

However, when the XSLT processor encounters the following code

```
<xsl:apply-templates select="MediaCompanies/Company"/>
```

the XPath expression

```
MediaCompanies/Company
```

in the select attribute of the <xsl:apply-templates> element is evaluated and a node set is returned. Given the source XML document, the returned node set would contain two element nodes, one for each of two <Company> elements in the source document.

For each of the nodes in the node set, the most closely matching <xsl:template> element is applied. In our XSLT stylesheet there is only one matching <xsl:template> element

```
<xsl:template match="Company">
<p><b><xsl:value-of select="."/></b> whose abbreviated name is
<xsl:value-of select="@abbreviation"/></p>
</xsl:template>
```

When that template for each of the two element nodes is instantiated, the context node is the element node which represents the <Company> element. Thus, the select attribute of the <xsl:value-of> element

```
<xsl:value-of select="."/>
```

selects the content of the context node itself (that is, a text node corresponding to the content of the <Company> element).

Similarly, the code

```
<xsl:value-of select="@abbreviation"/>
```

is evaluated with the same context node—the element node representing a <Company> element. Thus the value of the abbreviation attribute of the <Company> element is output to the result tree as a string, since @abbreviation is the abbreviated XPath syntax to select an abbreviation attribute node.

So, as you have seen, the <xsl:apply-templates> element changes the context node when it is applied. When the node set returned by the <xsl:apply-templates> element contains more than one node, then the context node changes when the template corresponding to each node is instantiated.

This idea of a context node is so important that I will walk you through another example, but in less detail so that you can be sure that you have grasped the idea. Every XPath expression or location path is evaluated relative to a context, so if you are fuzzy about what the context node is, you are also likely going to be fuzzy about what you expect the output of a transformation to be. That is a recipe for frustration and a lot of wasted time!

This example should be a little more testing for you, but the subject matter should be familiar. Listing 2.8 is a highly abbreviated, simplified personnel record.

If we want to display the content of the personnel record as HTML/XHTML, we can use an XSLT stylesheet like that shown in Listing 2.9.

```
<?xml version='1.0'?>
<PersonnelRecord>
  <Name>
    <FirstName>John</FirstName>
    <MiddleInitial>D.</MiddleInitial>
    <LastName>Smith</LastName>
  </Name>
  <Address>
    <Street>123 Any Street</Street>
    <Town>Anytown</Town>
    <State>New York</State>
  </Address>
  <PhoneNumbers>
    <PhoneNumber type="Work">123 333 3455</PhoneNumber>
    <PhoneNumber type="Mobile">234 432 1234</PhoneNumber>
    <PhoneNumber type="Home">987 876 5432</PhoneNumber>
  </PhoneNumbers>
</PersonnelRecord>
```

Listing 2.8 A Brief Personnel Record in XML (PersonnelRecord.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
  <xsl:apply-templates/>
</body>
</html>
</xsl:template>
```

Listing 2.9 An XSLT Stylesheet to Transform PersonnelRecord.xml to HTML (PersonnelRecord.xsl).

```
<xsl:template match="PersonnelRecord">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="Name">
<h2>Name:</h2>
 <xsl:apply-templates/>
</xsl:template>

<xsl:template match="FirstName">
<p><xsl:value-of select="."/></p>
</xsl:template>

<xsl:template match="MiddleInitial">
<p><xsl:value-of select="."/></p>
</xsl:template>

<xsl:template match="LastName">
<p><xsl:value-of select="."/></p>
</xsl:template>

<xsl:template match="Address">
<br /><h2>Address:</h2>
  <p><xsl:value-of select="Street"/></p>
  <p><xsl:value-of select="Town"/></p>
  <p><xsl:value-of select="State"/></p>
</xsl:template>

<xsl:template match="Street"/>

<xsl:template match="Town"/>

<xsl:template match="State"/>

<xsl:template match="PhoneNumbers">
<br /><h2>Phone Numbers</h2>
<xsl:apply-templates />
</xsl:template>

<xsl:template match="PhoneNumber">
<p><xsl:value-of select="@type"/><xsl:text>: </xsl:text><xsl:value-of
  select="."/></p>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.9 (*Continued*)

Note that many of the <xsl:value-of> elements have a select attribute with value of ".", meaning the context node. Thus the stylesheet, when considered together with its output HTML document, provides a useful way to monitor how the context node changes as the stylesheet is processed.

Let's follow the various changes in the context node. Again, we start with the root node as context node, as indicated by

```
<xsl:template match="/">
```

From within that template the context node changes due to the element:

```
<xsl:apply-templates />
```

In the absence of a select attribute the default selection comes into play, so that it is the child element nodes of the root node that are matched in the resulting node set. There is only one element node that is a child of the root node—the element node that represents the element root in the source document.

Thus that element node becomes the context node when we instantiate

```
<xsl:template match="PersonnelRecord">
```

Again, there is no select attribute on the <xsl:apply-templates> element contained in that template and so, once more by default, the child element nodes of the node that represents the <PersonnelRecord> element are in the node set returned by <xsl:apply-templates>. There are three nodes in that node set—the element nodes that represent the <Name>, the <Address>, and the <PhoneNumbers> elements. Each of those nodes is evaluated in turn.

If we look at the template that matches the Name element, then when the <xsl:template> element with match attribute equal to "Name" is accessed, the context node is the element node representing the Name element.

When we reach the <xsl:apply-templates/> element in that template, there is a node set containing three element nodes, representing the <FirstName>, the <MiddleInitial>, and the <LastName> elements. Then each of those nodes in turn becomes the context node. Thus, successively the child element nodes of the element node representing the <Name> element become the context node. Thus, for example, when the template for the FirstName element is accessed as follows, the context node becomes the node which represents the FirstName element.

```
<xsl:template match="FirstName">
```

Thus when the context node is the element node representing the <FirstName> element, to display the text content of the <FirstName> element, we can use the XPath expression '.', which is an expression in the relative abbreviated syntax, which simply means the context node itself (see Chapter 4, "The Four XPath Syntaxes," for details).

The remainder of the node set selected by the <xsl:apply-templates> element in the template for the element node representing the <Name> element is then processed.

Then the remainder of the node set selected by the <xsl:apply-templates> element in the template matched by the node representing the <PersonnelRecord> element are processed—the nodes representing the <Address> and <PhoneNumber> elements.

As you can see in Figure 2.3, the order of processing by the XSLT transformation successively outputs the value of many different context nodes.

Current Node

XSLT has a concept termed the *current node*. Often the XSLT current node is the same node as the XPath context node, but that is not always the case.

In the previous section I showed you that the context node is the node selected by the <xsl:apply-templates> element. I could also have said that that was the current node, since in that situation the current node and the context node are the same node. More broadly, the <xsl:apply-templates> elements return the current node list, which is an ordered node set.

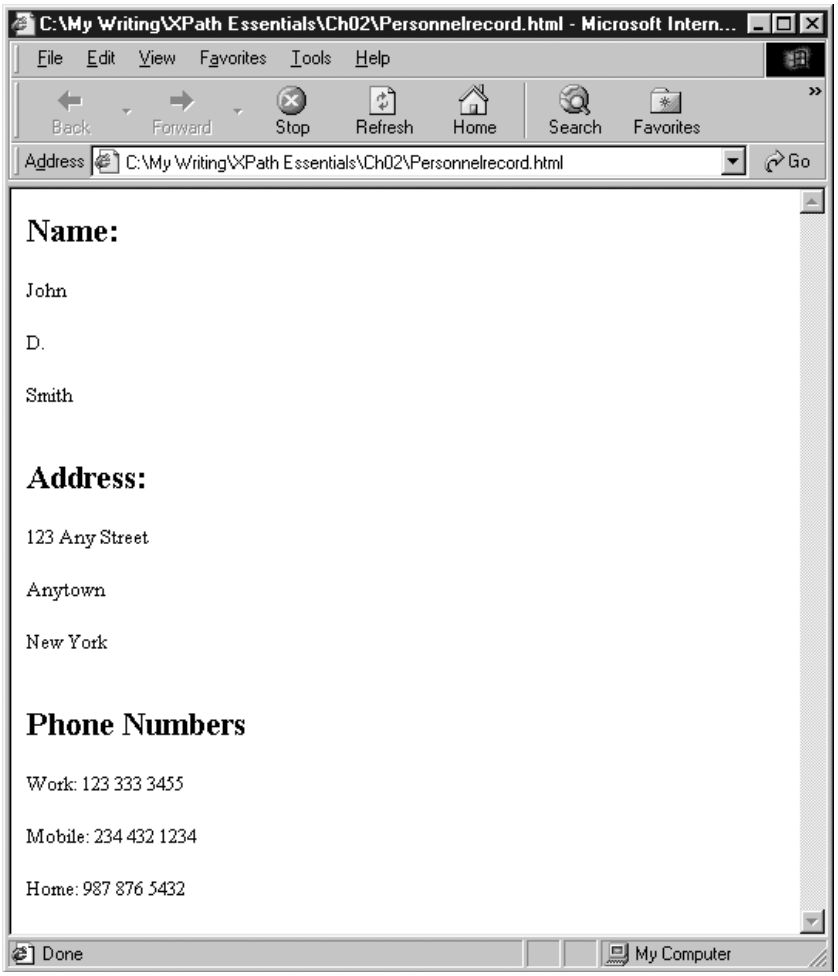


Figure 2.3 Output from the XSLT Transformation Reflecting Processing of Multiple Context Nodes.

NOTE If you are familiar with the mathematical notion of a set that is not ordered, you may find the idea of an ordered set a little confusing, partly because the XSLT specification seems to refer to it both as a "node set" and a "list." The terminology in the specification could be clearer, particularly since the concepts of current node and context node that relate to this are also potentially confusing.

Similarly in the processing of an <xsl:for-each> element, the current node and the context node are the same node.

The time when the current node and the context node are not the same is when a predicate in an XPath expression is being evaluated. The context node is the node then being tested by the predicate.

Having introduced the current node, I want to use it as the basis to further define the XPath context that I introduced in the previous section.

The current node is a node in the current node list. The position of the current node in the current node list gives the position of the context node.

The size of the current node list determines the context size.

Function Calls

A call to an XPath function may be part of an XPath expression. XPath functions are described later in this chapter. The core XPath function library, as well as the XPath extension functions, are described in detail in Chapter 5, "XPath Functions."

For example, the following XPath expression incorporates a call to the position ( ) function:

```
/child::book/child::chapter[position()>2]
```

This expression selects chapter elements for which the position ( ) function returns a value greater than two, which are also children of the book element, which is a child of the root node. When I put XPath expressions into words you can see that it can be pretty wordy. The XPath syntax is more succinct than English and is also more succinct than expressing similar information in XML would be. The detailed syntax of XPath expressions of this type will be described in the Location Paths section that follows.

Let's apply that location path to an example. If we had the source XML document shown in Listing 2.10 and we wanted to create HTML output only from the content of <chapter> elements after Chapter 2, we could use the stylesheet in Listing 2.11.

We use the location path as the value of the <xsl:apply-templates> element in the main template.

NOTE As far as I am aware there is no standard term in XSLT of "main template," but in this book I will use it to refer to the template where the value of the match attribute is equal to "/", in other words, the template which is instantiated with the root node as the context node. I am sure you will agree that "main template" is a more succinct term.

```
<?xml version='1.0'?>
<book>
<chapter number="1">First Chapter</chapter>
<chapter number="2">Second Chapter</chapter>
<chapter number="3">Third Chapter</chapter>
<chapter number="4">Fourth Chapter</chapter>
</book>
```

Listing 2.10 A Simple Book Structure in XML (book2.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the position() function</title>
</head>
<body>
<p>This is the text in the &lt;chapter&gt; elements after position 2:</p>
<xsl:apply-templates
select="/child::book/child::chapter[position()>2]"/>
</body>
</html>
</xsl:template>

<xsl:template match="/book/chapter">
<p><xsl:value-of select="."/></p>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.11 An XSLT Stylesheet to Selectively Output the Content of <chapter> Elements (book2.xsl).

So what does it mean? The first location step selects the child element node of the root node, which represents the element root <book> element in the source document.

```
child::book
```

The second location step first selects all element node children representing <chapter> elements as the node set to which the predicate is to be applied.

```
child::chapter[position()>2]
```

The predicate uses the XPath position ( ) function.

```
[position()>2]
```

The position ( ) function, not surprisingly, evaluates the context position of the context node within the context size. The position ( ) function is applied to the selected nodes in document order. The first chapter is in context position 1 and so is not selected for template instantiation by the <xsl:apply-templates> element since its context position is not greater than 2. The same result applied to the second element node, since again its context position is not greater than 2.

Only the third and fourth element nodes representing <chapter> elements are selected for processing by the expression contained in the select attribute of the <xsl:apply-templates> element since it is true of both nodes that their context position exceeds 2.

The output of the transformation is shown in Figure 2.4.

NOTE However, there are certain places where an XPath function call may not be placed. For example, the following expression

/child::book/child::chapter/position ( ) >2

would be illegal since an XPath function is not permitted immediately after a "/" character in an expression.

Let's move on to take a closer look at XPath location paths, since we have already been using them without my having explained to you in detail what they are.

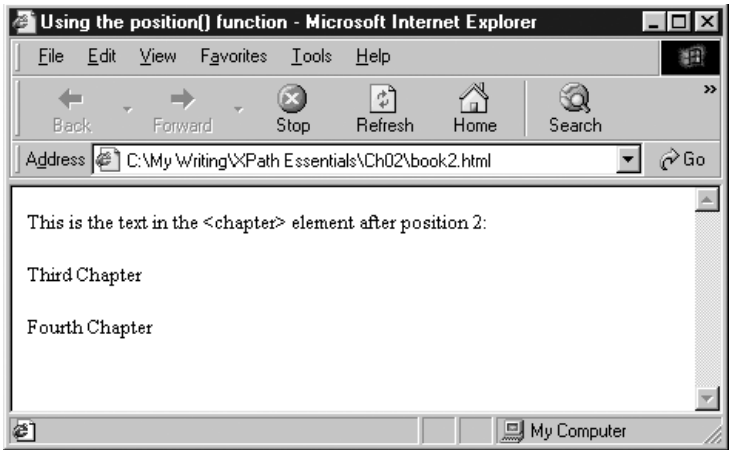

Using the position() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\book2.html
Go
This is the text in the <chapter> element after position 2:
Third Chapter
Fourth Chapter
My Computer

Figure 2.4 Display of the Content of <chapter> Elements for Which the position ( ) Function Returns More than 2.

XPath Location Paths

Not all XPath expressions are location paths. Only those expressions that return a node set are location paths.

Thus it is possible to correctly use the terms *expression* and *location path* interchangeably when referring to location paths, since all location paths are at the same time both a location path and an expression.

The XPath location path is by far the most important type of XPath expression. The greater part of this book will be devoted to the location path and understanding what it is, what its component parts are, and how to use the component parts in a variety of real-life programming situations.

Four syntaxes exist in XPath to describe location paths:

- Unabbreviated absolute location paths
- Unabbreviated relative location paths
- Abbreviated absolute location paths
- Abbreviated relative location paths

The four syntax forms are described in detail in "4, The Four XPath Syntaxes." The unabbreviated syntax may be used to describe any valid XPath location path. The abbreviated forms of the syntax provide a shorter form to express commonly used XPath location paths, but for some location paths there is no available abbreviated syntax. For occasions when you want to use such location paths, it will be necessary to use the unabbreviated forms of syntax.

So, in order to give you an impression of what these different syntaxes look like, let's create a couple of simple examples.

The first example will use each of the four syntaxes to select one element and output it in an HTML document. Listing 2.12 shows the source XML document.

Listing 2.13 is the XSLT stylesheet using an XPath location path, which is expressed in the unabbreviated absolute syntax.

The important part of the code for the purposes of illustrating the unabbreviated absolute syntax is the expression in the select attribute of the <xsl:apply-templates> element:

```
select="/child::AnnualReports/child::AnnualReport[attribute::Year='2001']"
```

```
<?xml version='1.0'?>
<AnnualReports>
<AnnualReport Year="2001">The 2001 annual report.</AnnualReport>
<AnnualReport Year="2002">The 2002 annual report.</AnnualReport>
<AnnualReport Year="2003">The 2003 annual report.</AnnualReport>
</AnnualReports>
```

Listing 2.12 A Catalog of Annual Reports in XML AnnualReports.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Four XPath Syntaxes Demo 1</title>
</head>
<body>
<h2>The Annual Report for XMML.com for the year 2001</h2>
<xsl:apply-templates
select="/child::AnnualReports/child::AnnualReport[attribute::Year='2001']
  "/>
</body>
</html>
</xsl:template>

<xsl:template match="AnnualReport">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.13 Stylesheet to Transform AnnualReports.xml (AnnualReports.xsl).

Let's dissect the location path. It starts with a "/" character indicating, since it starts with that character, that it is an absolute location path with the root node as its context node. Then the location step chooses element nodes in the child axis which represent <AnnualReports> elements. Of course there is only one, the element root of the source document.

```
child::AnnualReports
```

With that node as the context node, the next location step selects element nodes in the child axis which represent <AnnualReport> elements.

```
child::AnnualReport[attribute::Year='2001']
```

The predicate selects from among those element nodes only those element nodes where the attribute node which represents the Year attribute has the string value of "2001."

```
[attribute::Year='2001']
```

Note that the predicate refines which element nodes are selected on the basis of what value the attribute has. It does not select any attribute nodes.

When we run our stylesheet on the XML source document we get the HTML output on screen (see Figure 2.5), which is exactly what we wanted.

If we want to select exactly the same content, but using the unabbreviated relative syntax, we can do so by a simple modification of the select attribute of the <xsl:apply-templates> element, like this

```
<xsl:apply-templates select="child::AnnualReports/child::AnnualReport
  [attribute::Year='2001']"/>
```

You'll have to look twice to see the difference because it is very slight. The only difference is that the first "/" character is removed. Thus the expression is evaluated relative to the context node, which is determined by the template that the <xsl:apply-templates> element is contained in. In this case the <xsl:template> element has a match attribute with a value of "/", indicating that the context node is the root node.

Thus our context node remains unchanged but is arrived at by different methods. In the unabbreviated absolute syntax we declare the context node explicitly by having the "/" character as the first character in the location path. In the unabbreviated relative syntax, it is the <xsl:template> that determines the context node, and the location path is evaluated relative to the context node determined by the containing <xsl:template> element.

NOTE **I have indicated that the context node is determined by the match attribute of the <xsl:template> element. That is the case in these simple examples, but the context node may change several times during the evaluation of more complex templates than those demonstrated so far. So it is not safe to assume that the match attribute of the <xsl:template> element determines the context node throughout the template body and its associated applied and called templates.**

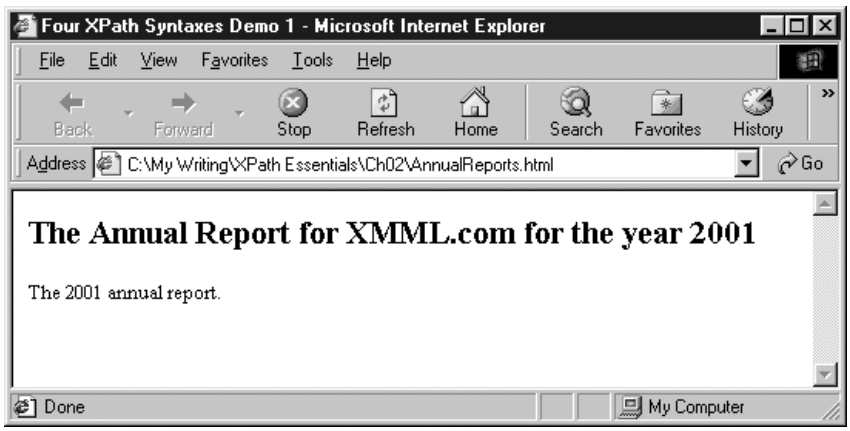

Four XPath Syntaxes Demo 1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address C:\My Writing\XPath Essentials\Ch02\AnnualReports.html
Go
The Annual Report for XMML.com for the year 2001
The 2001 annual report.
Done
My Computer

Figure 2.5 Screen Shot of HTML Output Using the Select Attribute.

Next let's create the same output by using the abbreviated absolute syntax. Here is the code for the <xsl:apply-templates> element.

```
<xsl:apply-templates select="/AnnualReports/AnnualReport[@Year='2001']"/>
```

As you can see, the expression is now significantly shorter. Abbreviated syntax means what it says! Let's break it down into its component parts. The first character is the "/" character indicating that we start with the root node as context node. Then the first location step means the same as `child::AnnualReports`.

```
AnnualReports
```

The child axis is the default axis, so it need not be explicitly expressed in the abbreviated syntax.

The second location step is

```
AnnualReport[@Year='2001']
```

which is equivalent to `child::AnnualReport[attribute::Year='2001']` since the child axis is again the default axis and the `@Year` is the abbreviated form of `attribute::Year`.

The abbreviated relative syntax, as you may already have guessed, is

```
<xsl:apply-templates select="AnnualReports/AnnualReport[@Year='2001']"/>
```

The only change from the abbreviated absolute syntax is the removal of the "/" character at the beginning of the location path.

Having seen the method of selecting an element using all four syntaxes, let's look at how to select an attribute. We will simply output the value of the Year attribute on the second <AnnualReport> element. We will use the same XML source document and modify the stylesheet.

Using the unabbreviated absolute syntax, the stylesheet looks like Listing 2.14.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Four XPath Syntaxes Demo 2</title>
</head>
<body>
<h2>
The annual report for XMML.com for the year
<xsl:value-of
```

Listing 2.14 Selecting Attributes (AnnualReports2.xsl).

```
select="/child::AnnualReports/child::AnnualReport[position()=2]/
  attribute::Year"/>
</h2>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.14 (*Continued*)

Again the meat of the XPath is in the select attribute of the <xsl:value-of> element:

```
<xsl:value-of  select="/child::AnnualReports/child::AnnualReport
  [position()=2]/attribute::Year"/>
```

The context node is the root node as before, indicated by the initial "/" character, and the first location step works as in the previous examples.

The second location step starts with the element node representing the <AnnualReports> element as context node; therefore, the element nodes which represent <AnnualReport> elements are selected.

```
child::AnnualReport[position()=2]
```

However, the predicate `[position()=2]` refines that selected node set so that only a single node is left—the element node representing the Annual Report for 2002.

With that single element node as the context node, the third location step is evaluated.

```
attribute::Year
```

The attribute axis, which selects only attribute nodes, is chosen. It is specified that only an attribute node that represents a Year attribute is selected.

Thus, putting all that together, the <xsl:value-of> element selects the value of the Year attribute of the second <AnnualReport> element for output to the result tree.

On screen our HTML output looks like Figure 2.6, confirming that the code has selected the correct attribute to display.

The unabbreviated relative syntax to produce the same output is

```
<xsl:value-of  select="child::AnnualReports/child::AnnualReport
  [position()=2]/attribute::Year"/>
```

As in the previous example, the only difference from the unabbreviated absolute syntax is the removal of the initial "/" character.

The abbreviated absolute syntax looks like this

```
<xsl:value-of  select="/AnnualReports/AnnualReport[2]/@Year"/>
```

Four XPath Syntaxes Demo 2 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address C:\My Writing\XPath Essentials\Ch02\AnnualReports2.html
Go
The annual report for XMML.com for the year 2002
Done
My Computer

Figure 2.6 The Year Attribute for the Second <AnnualReport> Element Is Displayed.

Starting with the root node as context node, the child axis is chosen by default with element nodes representing <AnnualReports> elements being selected in the first location step. In the second location step, again the child axis is chosen by default and the element nodes representing <AnnualReport> elements are chosen, but the node set is refined by applying the predicate `[2]` which is the abbreviated form for `[position()=2]`. The final location step, `@Year`, is the abbreviated form of `attribute::Year`.

The abbreviated relative syntax looks like this:

```
<xsl:value-of  select="AnnualReports/AnnualReport[2]/@Year"/>
```

The context node is determined by the containing main template, which has a match attribute of "/", meaning the root node. The location steps are thereafter the same in syntax and meaning as described for the abbreviated absolute syntax.

In these examples you have been introduced to the use of all four XPath syntaxes. The child axis, which by default selects element nodes, and the attribute axis are the most frequently used of the 13 XPath axes and can be expressed in all four syntaxes.

Some of the less commonly used axes cannot be expressed at all using either of the abbreviated syntaxes. In that situation you must use the appropriate unabbreviated syntax.

In these examples I have mentioned and analyzed example location steps several times without describing their structure in detail. It's now time to take a closer look at location steps.

Location Steps

Location steps occur within the setting of an XPath expression or location path. Remember, a location path is simply a subset of XPath expressions that returns a node set. XPath expressions that return a string, Boolean, or number are not location paths.

An XPath location path consists of one or more location steps.

The simplest location paths have only a single location step. If we had the source document shown in Listing 2.15, and we wanted to set the context node to the element node representing the <PartCatalog> element then within the main template, we could express that by the following

```
<xsl:apply-templates select="child::PartsCatalog"/>
```

In abbreviated relative syntax, it would be expressed as

```
<xsl:apply-templates select="PartsCatalog"/>
```

Assuming that we had a suitable <xsl:template> to match the <PartsCatalog> element, our desired processing would be carried out.

Each location step consists of

- An axis
- A node test
- An optional predicate

The XPath axis corresponds to the direction along which the hierarchy of nodes is being navigated. The node test determines whether any node found along the stipulated axis matches the type of node indicated. The optional predicate allows further filtering of the node set yielded when the node test has been applied in the specified axis.

In the previous example, the axis was the child axis—which was explicit in the unabbreviated relative syntax or implicit in the abbreviated relative syntax (since the child axis is the default axis)—and the node test was to test whether or not the name of the element node was "PartsCatalog." In that example no predicate was present.

Location paths may have multiple location steps. Thus, in our example if we wished to process only the second element node representing a <Part> element, we could do that using

```
<xsl:apply-templates select="child::PartsCatalog/child::Part
  [attribute::code='AB-234']"/>
```

or

```
<xsl:apply-templates select="PartsCatalog/Part[@code='AB-234']"/>
```

```
<?xml version='1.0'?>
<PartsCatalog>
<Part code="AB-123">Wonder widget</Part>
<Part code="AB-234">Great gadget</Part>
<Part code="AB-345">Sizzling something</Part>
</PartsCatalog>
```

Listing 2.15 A Simple Parts Catalog in XML. (PartsCatalog.xml)

In the unabbreviated form the first location step is

```
child::PartsCatalog
```

The second location step is

```
child::Part[attribute::code='AB-234']
```

with the "/" character being the separator between each location step. The "/" character is the separator for both the unabbreviated and abbreviated syntaxes.

I have mentioned the axis, node test, and predicate as part of a location step. In the sections that follow, we will examine in some detail the axes provided by XPath, node tests that can be applied, and some more example predicates.

Axes

XPath has a total of 13 different axes, which we will look at in this section. An XPath axis tells the XPath processor which "direction" to head in as it navigates around the hierarchical tree of nodes.

However, the axes can be more conceptual than visible. By that I mean that you can't simply follow the lines we might draw between nodes in a source tree of the type you saw earlier in this chapter and which are developed in more detail in Chapter 3, "XPath Data Model."

Here is a full list of the 13 different axes, with brief descriptions:

- **The child axis**—contains the children of the context node
- **The descendant axis**—contains the children of the context node, the children of those children, etc.
- **The parent axis**—contains the parent of the context node if it has one
- **The ancestor axis**—contains the ancestors of the context node, that is, the parent of the context node, its parent, etc., if it has one.
- **The following-sibling axis**—contains the following siblings of the context node
- **The preceding-sibling axis**—contains the preceding siblings of the context node
- **The following axis**—contains all nodes which occur after the context node, in document order
- **The preceding axis**—contains all nodes which occur before the context node, in document order
- **The attribute axis**—contains all the attribute nodes, if any, of the context node
- **The namespace axis**—contains all the namespace nodes, if any, of the context node
- **The self axis**—contains only the context node
- **The descendant-or-self**—contains the context node and its descendants
- **The ancestor-or-self**—contains the context node and its ancestors

NOTE **In the following section headers, which describe the XPath Axes, I have used uppercase letters, since the name of the axis was part of a section header. When using the unabbreviated syntax that uses the names of the axes explicitly it is important to use lowercase characters only.**

Child Axis

The child axis defines the children of the context node. I appreciate that this definition is slightly circular, so let's look more closely at what that means.

If we have a part of a source document as in Listing 2.16, the element nodes that represent the <Bikini> element and the <SwimTrunks> element are child nodes of the element node that represents the <BeachWear> element.

However, if our source document were a little more detailed, like this

```
<BeachWear>
Be sure to look carefully at our Summer 2002 catalog.
<!-- Update weekly according to stocks in hand. -->
<Bikini>Striking pink, sure to catch the eye</Bikini>
<SwimTrunks>Unforgettable - in fluorescent lime green</SwimTrunks>
</BeachWear>
```

the text node that contains "Be sure to look carefully at our Summer 2002 catalog." and the comment node that contains the reminder "Update weekly according to stocks in hand." are also child nodes of the element node that represents the <BeachWear> element.

In fact the nodes selected in the child axis may be element nodes or text nodes or comment nodes as I have just mentioned. In addition the child axis can select processing instruction nodes. However, namespace nodes and attribute nodes are not child nodes of the element node with which they are associated. Namespace nodes and attribute nodes are accessed via separate axes—the namespace axis and the attribute axis, respectively.

The *principal node type* of the child axis is the element node. Thus if we use the location paths `child::node()` and `child::*` we get different node sets selected. The `child::node()` location path returns all nodes in the child axis, that is, any element, comment, text, and processing instruction nodes, whereas the `child::*` location path returns only the element nodes that are child nodes, that is, nodes which are of the principal node type. The following stylesheet, Listing 2.17, shows this in action.

```
<BeachWear>
<Bikini>Striking pink, sure to catch the eye</Bikini>
<SwimTrunks>Unforgettable - in fluorescent lime green</SwimTrunks>
</BeachWear>
```

Listing 2.16 A Beachwear List in XML (BeachWear.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Child axis - selecting node types</title>
</head>
<body>
<xsl:apply-templates select="BeachWear"/>
</body>
</html>
</xsl:template>

<xsl:template match="BeachWear">
<h3>Here are child nodes selected by the location path
"child::node()".</h3>
<xsl:for-each select="child::node()">
<p><xsl:value-of select="."/></p>
</xsl:for-each>
<h3>Here are child nodes selected by the location path "child::*".</h3>
<xsl:for-each select="child::*">
<p><xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.17 The XSLT stylesheet to Transform Beachwear.xml (Beachwear.xsl).

As you can see in the output in Figure 2.7, the `child::node()` location path selects child nodes of all node types in document order. Thus, for our example, the content of the text node, the comment node, and both element nodes is displayed. The `child::*` location path selects only element node children, in our example the content of the <Bikini> and <SwimTrunks> elements.

Now let's move on and look at a related axis (sorry for the pun)—the descendant axis.

Descendant Axis

The descendant axis selects the children of a context node, and for each of those children their child nodes are selected, and for each of those child nodes their children are selected, and so on. Thus the descendant axis can be viewed as applying the child axis recursively.

When using the absolute forms of the syntax, the descendant axis is effectively the same as saying select all nodes of a particular type that exist in the document. Thus if we had a document Listing 2.18 and we apply a stylesheet as in Listing 2.19, then all element

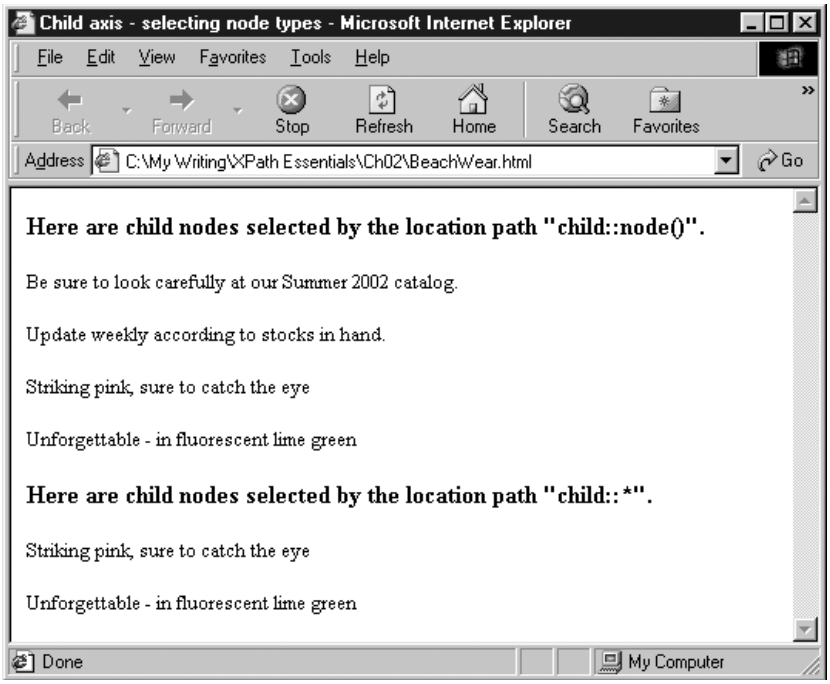

Child axis - selecting node types - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\BeachWear.html
Go
Here are child nodes selected by the location path "child::node()".
Be sure to look carefully at our Summer 2002 catalog.
Update weekly according to stocks in hand.
Striking pink, sure to catch the eye
Unforgettable - in fluorescent lime green
Here are child nodes selected by the location path "child::*".
Striking pink, sure to catch the eye
Unforgettable - in fluorescent lime green
Done
My Computer

Figure 2.7 The Difference in Meaning between child::node ( ) and child::* location Paths.

nodes which represent a <Section> element are selected. According to the <xsl:template match="Section"> element, the content of those element nodes is output to the HTML, as you can see in Figure 2.8.

```
<?xml version='1.0'?>
<Tome>
<Introduction>
<Section>A section.</Section>
<Section>Another section in the introduction</Section>
</Introduction>
<Chapter>
<Section>A section in a chapter.</Section>
<Section>Another section in a chapter.</Section>
</Chapter>
<Appendix>
<Section>A section in an appendix.</Section>
</Appendix>
</Tome>
```

Listing 2.18 An XML Description of a Book (Tome.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the descendant axis</title>
</head>
<body>
<h3>The location path /descendant::Section chooses all sections in the
document</h3>
<xsl:apply-templates select="/descendant::Section"/>
</body>
</html>
</xsl:template>

<xsl:template match="Section">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.19 Stylesheet to Transform Tome.xml (Tome.xsl).

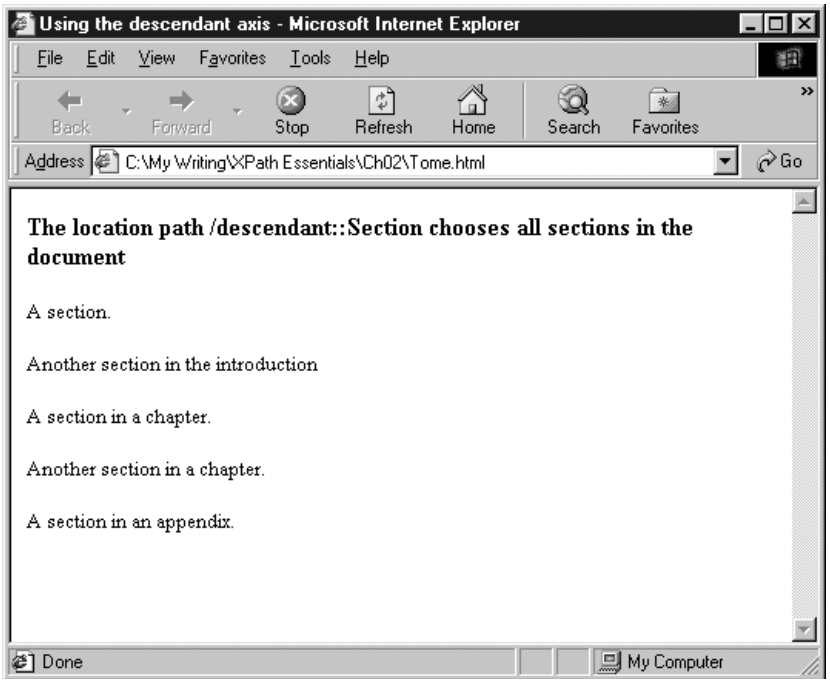

Using the descendant axis - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Tome.html
Go
The location path /descendant::Section chooses all sections in the document
A section.
Another section in the introduction
A section in a chapter.
Another section in a chapter.
A section in an appendix.
Done
My Computer

Figure 2.8 Using the Descendant Axis.

The descendant axis can be expressed in the abbreviated forms of the XPath syntax. Thus, using the abbreviated absolute location path syntax, the code `//Section` is equivalent to `/descendant::Section` in the unabbreviated absolute syntax.

The descendant axis can also be used (the second or later location steps) in a location path. For example, if we wanted to select all the <paragraph> elements in a publication but didn't know at which level they occurred, we could do so using a location path as follows assuming that the <publication> element was the element root of the document.

```
/publication//paragraph
```

In unabbreviated syntax that location path would appear as

```
/child::publication/descendant::paragraph
```

Note that a child element node is the only type of child node that may itself only have child element nodes. Comment nodes, text nodes, and processing instruction nodes are child nodes but cannot have child nodes of their own.

Having looked at the child and descendant axes, let's move on to examine the parent axis.

Parent Axis

The parent axis contains only a maximum of one node. The parent node may be either the root node or an element node.

The root node has no parent; therefore, when the context node is the root node, the parent axis is empty. For all other element nodes the parent axis contains one node. If the element node is the node representing the element root of the document, then its parent is the root node. For all other element nodes in a source document, the parent node is another element node.

The parent node is accessed using this syntax

```
parent::node()
```

in the unabbreviated syntax or

```
..
```

in the abbreviated syntax. Note that the abbreviated syntax is the same as is used for a parent directory in some operating systems.

Ancestor Axis

The ancestor axis selects all the ancestor nodes of the context node. This will return a node set that will have one root node member and zero or more element node members. The only exception is when the root node is the context node when the selected node set will be empty since the root node has neither a parent node nor any ancestor nodes.

Let's look at an example (see Listing 2.20).

```
<?xml version='1.0'?>
<Booklet>
<Chapter>
<Section>A first section</Section>
<Paragraph>A paragraph in the first section.</Paragraph>
</Chapter>
</Booklet>
```

Listing 2.20 A Simple Booklet in XML (Booklet.xml).

If we wished to select all the element nodes that were ancestors of the <Paragraph> element, we could do so like this

```
ancestor::node()
```

Note that there is no way to express the ancestor axis in abbreviated syntax.

Following-Sibling Axis

The following-sibling axis selects those nodes that are siblings of the context node (that is, the context node and its sibling nodes share a parent node) and which occur later in document order than the context node. The following-sibling axis is empty except when both the context node and any sibling nodes are element nodes.

If we have a source document as in Listing 2.21, we can select the third line item as the context node and then output the content of its siblings, which are in the following-sibling axis using the stylesheet in Listing 2.22.

If you compare the screen shot of the output (see Figure 2.9), you will see that it is the content of the <LineItem> elements in positions 4 and 5 (that is, the following siblings of the element node in position 3 which are output).

```
<?xml version='1.0'?>
<PurchaseOrder>
<Customer>Wonder Gadget Company Inc</Customer>
<LineItem>A widget</LineItem>
<LineItem>A dooda</LineItem>
<LineItem>A whotsit</LineItem>
<LineItem quantity="3">thingy</LineItem>
<LineItem quantity="4">gadget</LineItem>
</PurchaseOrder>
```

Listing 2.21 A Purchase Order in XML (PurchaseOrder.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>The following-sibling axis</title>
</head>
<body>
<h3>Applying the XPath following-sibling axis</h3>
<h4>The line items in the following sibling axis are:</h4>
<xsl:apply-templates
select="PurchaseOrder/LineItem[position()=3]/following-sibling::*"/>
</body>
</html>
</xsl:template>

<xsl:template match="LineItem">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.22 A Stylesheet to Demonstrate the Following-Sibling Axis (PurchaseOrder.xsl).

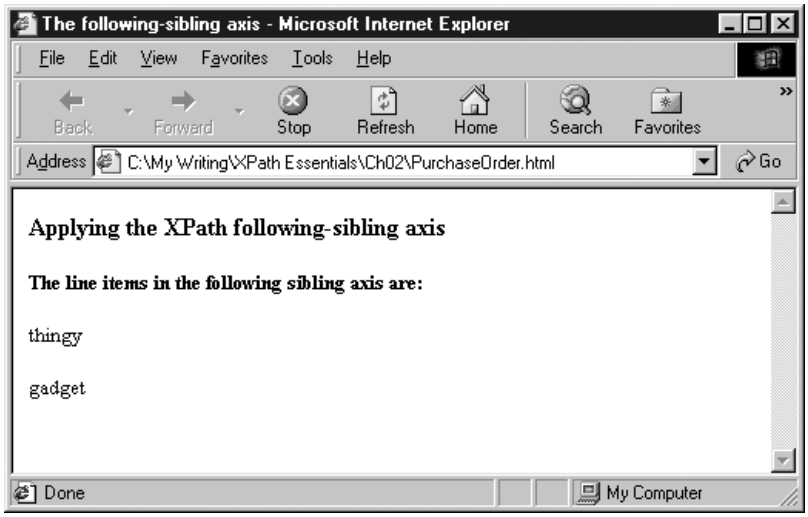

The following-sibling axis - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\PurchaseOrder.html
Go
Applying the XPath following-sibling axis
The line items in the following sibling axis are:
thingy
gadget
Done
My Computer

Figure 2.9 Use of the Following-Sibling Axis.

The following-sibling axis cannot be expressed in the abbreviated syntax; if you want to use that functionality, you must use unabbreviated syntax to express it.

Preceding-Sibling Axis

The preceding-sibling axis selects those nodes which are siblings of the context node (that is, the context node and its sibling nodes share a parent node) and which occur earlier in document order than the context node. The preceding-sibling axis is empty except when both the context node and any sibling nodes are element nodes.

To demonstrate the preceding-sibling axis in action, we need only modify the <xsl:apply-templates> element in the preceding example to read like this

```
<xsl:apply-templates select="PurchaseOrder/LineItem[position()=3]/
  preceding-sibling::*"/>
```

The output is shown in Figure 2.10 (the two preceding siblings of the context node that was in position 3).

Notice in Figure 2.10 that the content of the <Customer> element is also output on screen. Similar to the <LineItem> elements, the <Customer> element is also a child of the <PurchaseOrder> element. If we wanted to restrict the output only to the content of <LineItem> elements, we could modify the location path in the select attribute, as here.

```
<xsl:apply-templates select="PurchaseOrder/LineItem[position()=3]/
  preceding-sibling::LineItem"/>
```

The preceding-sibling axis cannot be expressed in the abbreviated syntax; if you want to use that functionality, you must use unabbreviated syntax to express it.

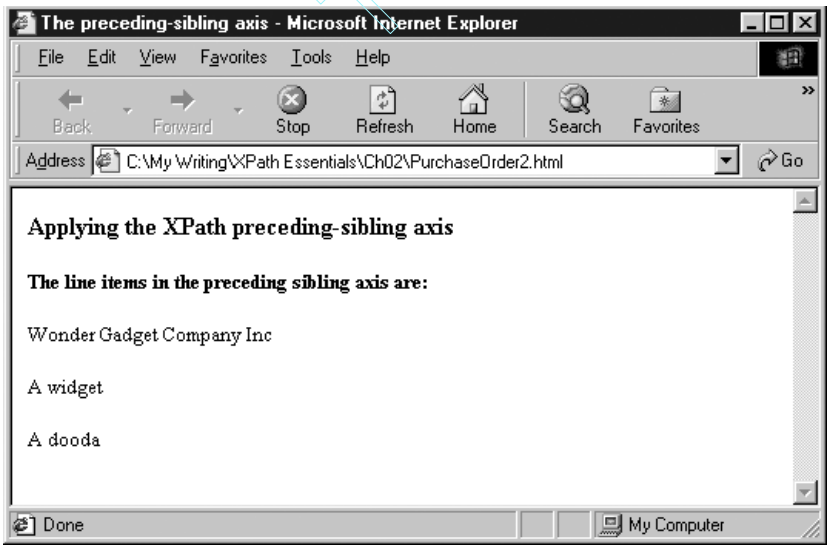

The preceding-sibling axis - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\PurchaseOrder2.html
Go
Applying the XPath preceding-sibling axis
The line items in the preceding sibling axis are:
Wonder Gadget Company Inc
A widget
A dooda
Done
My Computer

Figure 2.10 Using the Preceding-Sibling Axis.

Following Axis

The XPath specification defines the following axis in this way, "The following axis contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes." Quite a mouthful.

We will need a fairly long example document (see Listing 2.23) to demonstrate the following axis in action.

The stylesheet shown in Listing 2.24 sets the context node as the element node, representing the <Chapter> element that has the number attribute with value of "3."

I have included <xsl:template> elements to match <Appendix>, <Section>, and <Chapter> elements in the stylesheet so that when we come to test the following axis with a different context node we need only modify the <xsl:apply-templates> element.

```
<?xml version='1.0'?>
<Volume>
<Number>3</Number>
<Author>John Smith</Author>
<Title>Space Travel for dinosaurs</Title>
<PublicationDate>20th June 2303</PublicationDate>
<Introduction>You didn't know dinosaurs were still alive, did you?
  </Introduction>
<Chapter number="1">The first chapter starts here.
<Section>The first section in the first chapter.</Section>
<Section>The second section in the first chapter.</Section>
<Section>The third section in the third chapter.</Section>
</Chapter>
<Chapter number="2">The second chapter starts here.
<Section>The first section in the second chapter.</Section>
<Section>The second section in the second chapter.</Section>
<Section>The third section in the second chapter.</Section>
</Chapter>
<Chapter number="3">The third chapter starts here.
<Section>The first section in the third chapter.</Section>
<Section>The second section in the third chapter.</Section>
<Section>The third section in the third chapter.</Section>
</Chapter>
<Appendix>
<Section>
In this appendix we celebrate the first dinosaur in space in 2003.
<Section>
</Appendix>
</Volume>
```

Listing 2.23 A Description of a Book Expressed in XML (Volume.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>The XPath following axis</title>
</head>
<body>
<h3>Applying the XPath following axis</h3>
<h4>The elements in the following axis are:</h4>
<xsl:apply-templates
select="/Volume/Chapter[@number='3']/following::*"/>
</body>
</html>
</xsl:template>

<xsl:template match="Appendix">
<p><xsl:value-of select="name()"/></p>
</xsl:template>

<xsl:template match="Section">
<p><xsl:value-of select="name()"/></p>
</xsl:template>

<xsl:template match="Chapter">
<p><xsl:value-of select="name()"/></p>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.24 Stylesheet to Select Nodes in the Following Axis (Volume.xsl).

So, our context node in the above XSLT stylesheet is the element node representing the <Chapter> element with number attribute of value "3." The <xsl:apply-templates> element in the main template

```
<xsl:apply-templates select="/Volume/Chapter[@number='3']/following::*"/>
```

selects all nodes which are in the following axis and which are of the principal node type (the default node type) for the following axis, that is, element nodes.

You may need to look back at the definition of the following axis at the beginning of this section. We start with the context node as described. The element nodes representing <Section> elements that are nested within the third <Chapter> element are excluded since, although they follow the <Chapter> element in document order, they are descendants of the <Chapter> element and therefore belong to the child and descendant

axes, not the following axis. However, when we come to the nodes representing the <Appendix> and its nested <Section> nodes, they satisfy the criteria for being in the following axis. They come after the third <Chapter> element in document order, are not descendants of the third <Chapter> element, and are also not attribute or namespace nodes.

Thus when we come to view the output of the transformation, as in Figure 2.11, we see that the names of the <Appendix> element and the <Section> element are output.

If we alter the context node by modifying the <xsl:apply-templates> element in the main template as follows, we find that none of its child <Section> nodes are output but the name of the third <Chapter> node is output, together with the name of each of its <Section> element nodes as well as the names of the <Appendix> element node and of its <Section> element node child.

```
<xsl:apply-templates select="/Volume/Chapter[@number='2']/following::*"/>
```

I hope you can follow why each of these names was output in the way you can see in Figure 2.12. If not, then review the full explanation of the previous example and the definition at the beginning of this section.

Note that the following axis cannot be expressed in the abbreviated syntax and if you want to use that functionality you must use unabbreviated syntax to express it.

Preceding Axis

The XPath specification defines the preceding axis as follows, "The preceding axis contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes."

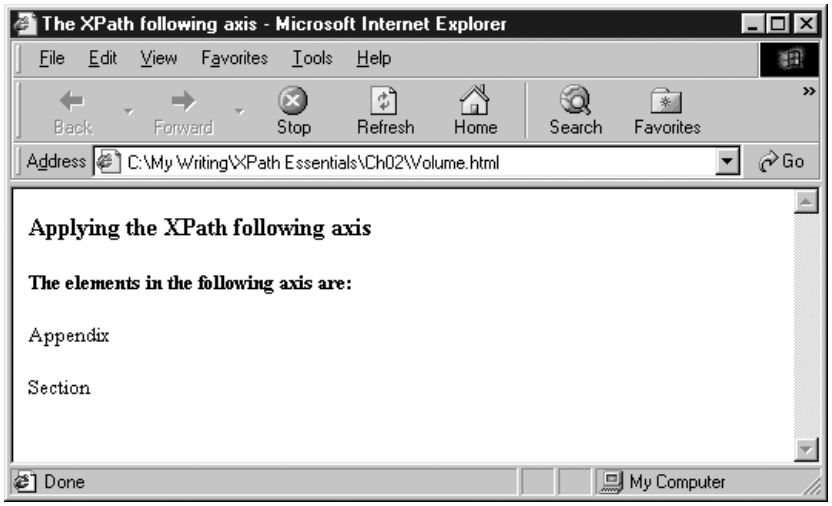


Figure 2.11 Using the XPath Following Axis.

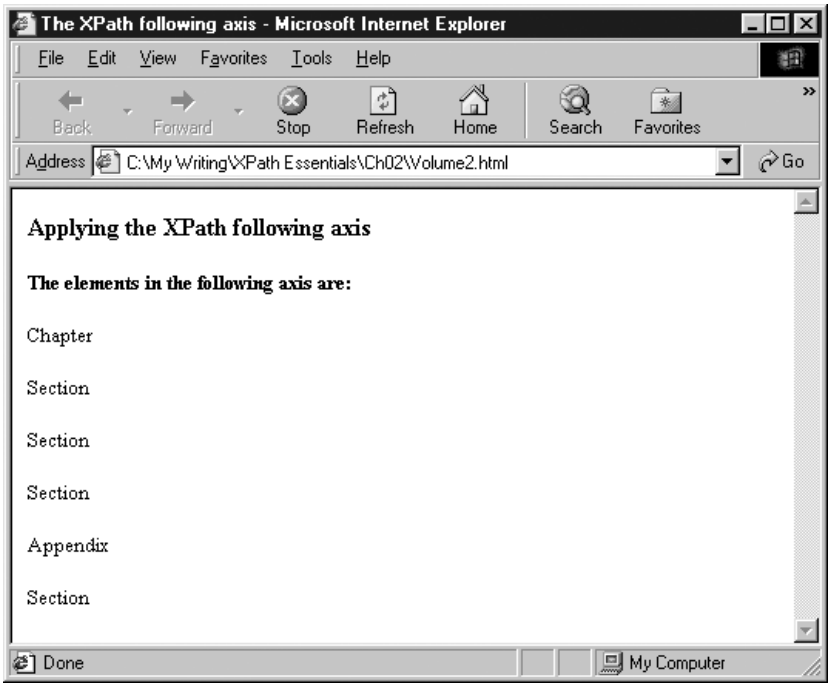


Figure 2.12 A Further Example Using the XPath Following Axis.

In the example to demonstrate the preceding axis in action we will use the same source document as in the examples for the following axis. The stylesheet is shown in Listing 2.25.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>The XPath preceding axis</title>
</head>
<body>
<h3>Applying the XPath preceding axis</h3>
<h4>The elements in the preceding axis are:</h4>
<xsl:apply-templates
select="/Volume/Chapter[@number='2']/preceding::*"/>
```

Listing 2.25 A Stylesheet to Demonstrate the XPath Preceding Axis (Volume3.xsl).

```
</body>
</html>
</xsl:template>

<xsl:template match="Appendix">
<p><xsl:value-of select="name()"/></p>
</xsl:template>

<xsl:template match="Section">
<p><xsl:value-of select="name()"/></p>
</xsl:template>

<xsl:template match="Chapter | Introduction | PublicationDate | Title |
Author | Number | Volume">
<p><xsl:value-of select="name()"/></p>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.25 (*Continued*)

The <xsl:apply-templates> element in the main template is what makes use of the preceding axis:

```
<xsl:apply-templates select="/Volume/Chapter[@number='2']/preceding::*"/>
```

The element node representing the second <Chapter> element is the context node. Thus we would expect all element nodes in the document that come earlier in document order to be output in the HTML, with the exception of the <Volume> element, since it is a parent node of the second <Chapter> node and therefore is in the parent and ancestor axes, not in the preceding axis. As you can see in Figure 2.13, all the element nodes which precede the second <Chapter> element are output except the name of the <Volume> node, despite the presence of a matching <xsl:template> element in the stylesheet.

Note that the preceding axis cannot be expressed in the abbreviated syntax; if you want to use that functionality, you must use unabbreviated syntax to express it.

Attribute Axis

The attribute axis contains the nodes associated with another node. Unless the other node is an element node, the attribute axis is empty.

You have seen the use of the attribute axis in earlier examples. In unabbreviated syntax you use the form `attribute::AttributeName` and in the abbreviated syntax the form `@AttributeName` to select a particular attribute.

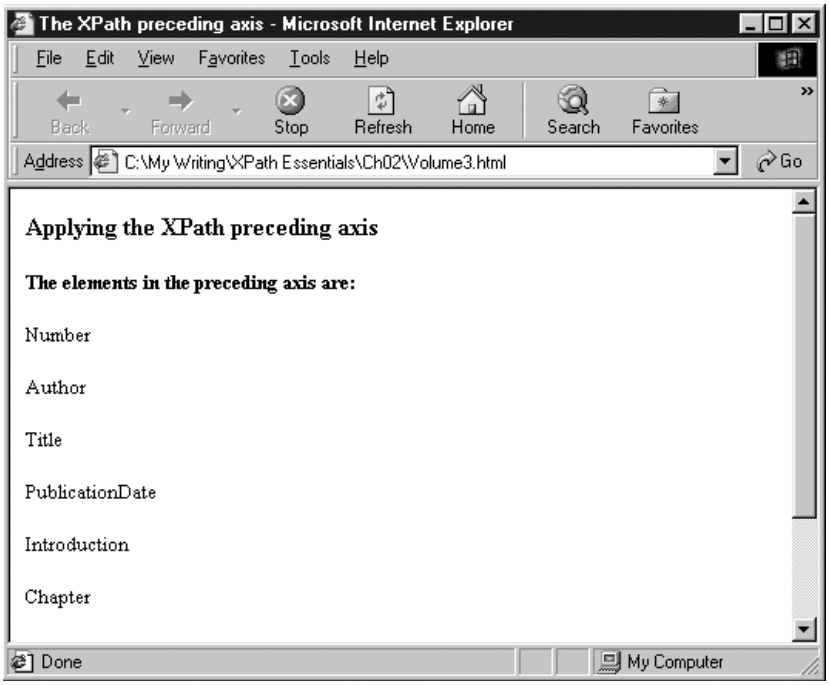


Figure 2.13 Output in the Preceding Axis.

Similarly if you want to select all attributes belonging to an element node then, in unabbreviated syntax, use the form `attribute::*` or in abbreviated syntax the form `@*` to select all attribute nodes.

The attribute axis can be expressed in both unabbreviated and abbreviated syntax.

Namespace Axis

The namespace axis contains the namespace nodes associated with the context node. Unless the context node is an element node, the namespace axis is empty.

Namespace nodes are not, in XPath terminology, children of the context node although the context node is called the parent of the namespace node.

Listing 2.26 is an example source document, including a single namespace declaration.

```
<?xml version='1.0'?>
<xmml:book xmlns:xmml="http://www.xmml.com/writing/">
<xmml:chapter>A first chapter</xmml:chapter>
<xmml:chapter>A second chapter</xmml:chapter>
</xmml:book>
```

Listing 2.26 A Book in XML with Namespace Prefixes (Namespace.xml).

Listing 2.27 is a stylesheet that uses the <xsl:apply-templates> element in the main template to cause the <xsl:template>, which matches the <xmml:book> element, to be instantiated. Within the latter template the <xsl:for-each> element causes the value of each namespace node associated with the element node representing the <xmml:book> element to be output on screen. Notice that the namespace declaration that follows is present in Listing 2.27, as well as in the source document.

```
xmlns:xmml="http://www.xmml.com/writing/"
```

This enables the correct namespace URI to be associated with any output element where that namespace URI is in scope.

Figure 2.14 shows the output. You can probably immediately see where the line "http://www.xmml.com/writing/" comes from, but where does "http://www.w3.org/XML/1998/namespace" come from?

It relates to the reserved prefix "xml", which is bound to http://www.w3.org/XML/1998/namespace. Thus all XML documents that make use of XML namespaces are implicitly associated with that namespace URI.

The namespace axis cannot be expressed in the abbreviated syntax; if you want to use that functionality, you must use unabbreviated syntax to express it.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xmml="http://www.xmml.com/writing/">

<xsl:template match="/">
<html>
<head>
<title>Namespace nodes in XPath</title>
</head>
<h3>Here are the namespace nodes in scope on the &lt;xmml:book&gt;
  element.</h3>
<xsl:apply-templates select="xmml:book"/>
</html>
</xsl:template>

<xsl:template match="xmml:book">
<xsl:for-each select="namespace::*">
<p><xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.27 Stylesheet to Transform xmmlbook.xml (Namespace.xsl).

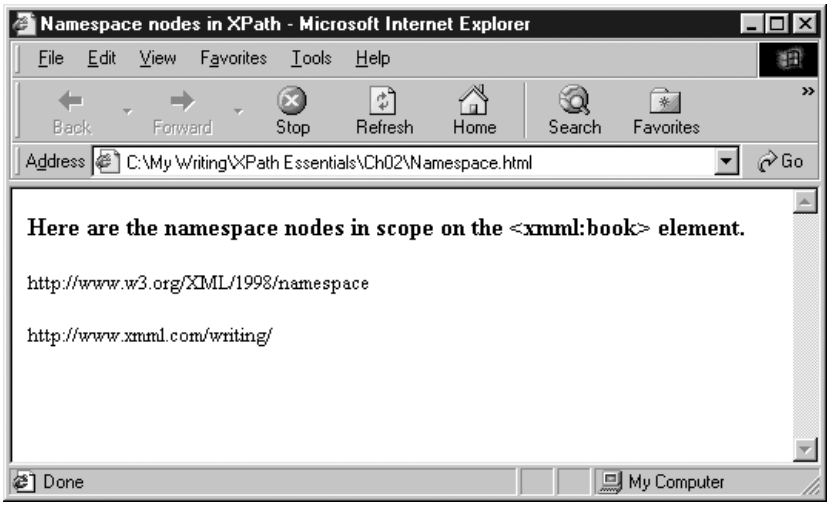

Namespace nodes in XPath - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Namespace.html
Go
Here are the namespace nodes in scope on the <xmml:book> element.
http://www.w3.org/XML/1998/namespace
http://www.xmml.com/writing/
Done
My Computer

Figure 2.14 A Demonstration of the Namespace Axis.

Self Axis

The self axis contains a single node—the context node.

In unabbreviated syntax it is expressed as

```
self::node()
```

In abbreviated syntax it is expressed as

```
.
```

Descendant-Or-Self Axis

The descendant-or-self axis includes the self axis and the descendant axis described earlier.

Ancestor-Or-Self Axis

The ancestor-or-self axis includes the self axis and the ancestor axis described earlier.

Having looked in some detail at the axis part of a location step, let's take a closer look at the node test.

Node Tests

Each XPath location path has at least one location step. And each location step must have a node test.

The node test can be viewed as a filter on the node set selected by the axis which forms the first part of the location step. I mentioned earlier that every axis had its own principal node type, which is effectively its default node type. Thus the syntax below selects only the principal node type for the axis in question.

```
AxisName::*
```

The principal node type for each axis type is listed in Table 2.1.

Thus, except when the axis is the attribute or namespace axis, the principal node type is the element node.

For a node test that is a QName to succeed, the node must be of the principal node type for the axis. Thus if we wanted to select the element nodes representing <Paragraph> children of the context node then we use the syntax

```
child::Paragraph
```

In that code "Paragraph" is a QName—with no namespace prefix. Thus we can only choose named element nodes in axes where the element node is the principal node type.

Similarly in the attribute axis we can only use the QName of the attribute to refine the selection of a node set in the attribute axis. Thus `attribute::type` selects the attribute node representing a type attribute present on the context node, assuming that the context node is an element node.

Table 2.1 Principal Node Types

AXIS	PRINCIPAL NODE TYPE
Child	Element
Descendant	Element
Parent	Element
Ancestor	Element
Following-sibling	Element
Preceding-sibling	Element
Following	Element
Preceding	Element
Attribute	Attribute
Namespace	Namespace
Self	Element
Descendant-or-self	Element
Ancestor-or-self	Element

This predilection for the principal node type when a QName is used means that a QName cannot, for example, be used to reference a comment node or a processing instruction node in the child axis. We can use the syntax

```
child::node()
```

to select all child nodes whatever their node type.

Alternatively, we can use

```
child::text()
```

to select all text nodes, or

```
child::comment()
```

to select all comment nodes, or

```
child::processing-instruction(Name)
```

to choose a processing instruction that has the target "Name".

A node test can also take the form of

```
NCName:*
```

Thus in the child axis it take the form of

```
child::NCName:*
```

What this means is that the child axis is chosen. The element node chosen has the QName with a namespace prefix that is an NCName. An NCName is a legal XML name except that an additional restriction applies—an NCName may not have a colon character within its name. That is a sensible restriction since it would otherwise be very difficult to parse a name like `child::elem:ent:this`.

It could mean an element node with namespace prefix `elem` and local part `ent:this` or it could mean an element node with namespace prefix `elem:ent` and local part `this`. By restricting both the namespace prefix and the local part to NCNames, that ambiguity and potential for error is avoided.

Predicates

A predicate is optional in any location step. A predicate acts as a further filter on the node set selected by the axis and node test. The axis is the first filter, the node test the second, and the predicate can add a third filter within a single location step.

There is, of course, nothing to prevent a predicate occurring in more than one location step in a location path, nor is there anything to prevent more than one predicate from being present within a single location step.

When trying to understand some predicates it is important to have grasped the notions of forward and reverse axes. An axis that contains only the context node and nodes that come after the context node in document order is called a *forward axis*. An axis that contains only the context node and nodes that come before the context node

TEAMFLY

is called a *reverse axis*. See Table 2.2 for details as to whether particular axes are forward or reverse axes or neither.

The self axis can be viewed as either a forward or reverse axis. In practice it makes no difference since the self axis only ever contains one node—the context node.

In addition to this notion of forward and reverse axes we also have to look at a notion called the *proximity position*—which basically means the order in which a node will be encountered when moving along an axis.

It might help to think of standing at a street junction. The first house you encounter going east has its proximity calculated in relation to the direction you are traveling. When you go west you are moving in reverse order by all the houses on the street and the first house is relative to the direction you are moving—which in this case is west. Similarly in XPath the proximity position relates to the direction of movement along a particular axis.

For example, we have a set of six chapters in a booklet, corresponding to the source document shown in Listing 2.28.

We can create a stylesheet (see Listing 2.29) that selects the fourth chapter as the pattern to match in the select attribute of the <xsl:apply-templates> element.

Let's take this step by step. First, notice that the <xsl:apply-templates> element in the main template provides a pattern

```
<xsl:apply-templates select="/Booklet/Chapter[position()=4]"/>
```

matched in the line

```
<xsl:template match="Chapter">
```

Table 2.2 Forward and Reverse XPath Axes

AXIS	DIRECTION
Child	Forward
Descendant	Forward
Parent	Reverse
Ancestor	Reverse
Following-sibling	Forward
Preceding-sibling	Reverse
Following	Forward
Preceding	Reverse
Attribute	
Namespace	
Self	Forward or reverse
Descendant-or-self	Forward
Ancestor-or-self	Reverse

```
<?xml version='1.0'?>
<Booklet>
<Chapter order="first">This is the first chapter</Chapter>
<Chapter order="second">This is the second chapter</Chapter>
<Chapter order="third">This is the third chapter</Chapter>
<Chapter order="fourth">This is the fourth chapter</Chapter>
<Chapter order="fifth">This is the fifth chapter</Chapter>
<Chapter order="sixth">This is the sixth chapter</Chapter>
</Booklet>
```

Listing 2.28 A Booklet Represented in XML (Booklet2.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Demo of proximity position in forward and reverse axes</title>
</head>
<body>
<h3>Demo of proximity position in forward and reverse axes</h3>
<xsl:apply-templates select="/Booklet/Chapter[position()=4]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Chapter">
<p>This is the content of this node: <xsl:value-of select="."/>.</p>
<h3>The following-sibling axis is a forward axis.</h3>
<p>Therefore the first node encountered has this content: "<xsl:value-of
select="following-sibling::*[position()=1]"/>"</p>
<br />
<h3>The preceding-sibling axis is a reverse axis.</h3>
<p>Therefore the first node encountered has this content: "<xsl:value-of
select="preceding-sibling::*[position()=1]"/>"</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.29 XSLT Stylesheet to Transform Booklet2.xml (Booklet2.xsl).

Thus when that template is instantiated, the context node is the element node that represents the <Chapter> element with the order attribute with value of "fourth." We can demonstrate the truth of that by the code

```
<p>This is the content of this node: <xsl:value-of select="."/>.</p>
```

which prints out the content of the context node. As you can see in Figure 2.15, the content of the context node is "This is the fourth chapter."

Next we choose to display the content of the first node encountered traveling along the following-sibling axis, which is a forward axis.

```
<xsl:value-of
select="following-sibling::*[position()=1]"/>
```

This causes the content of the fifth <Chapter> element to be printed out which is what we would expect. The following-sibling axis is a forward axis; thus we should expect the node in position 1 to be the element node immediately after the context node in document order.

Similarly, the code

```
<xsl:value-of
select="preceding-sibling::*[position()=1]"/>
```

causes the content of the first element node encountered while traveling along the preceding-sibling axis to be output. Since the preceding axis is a reverse axis, we would

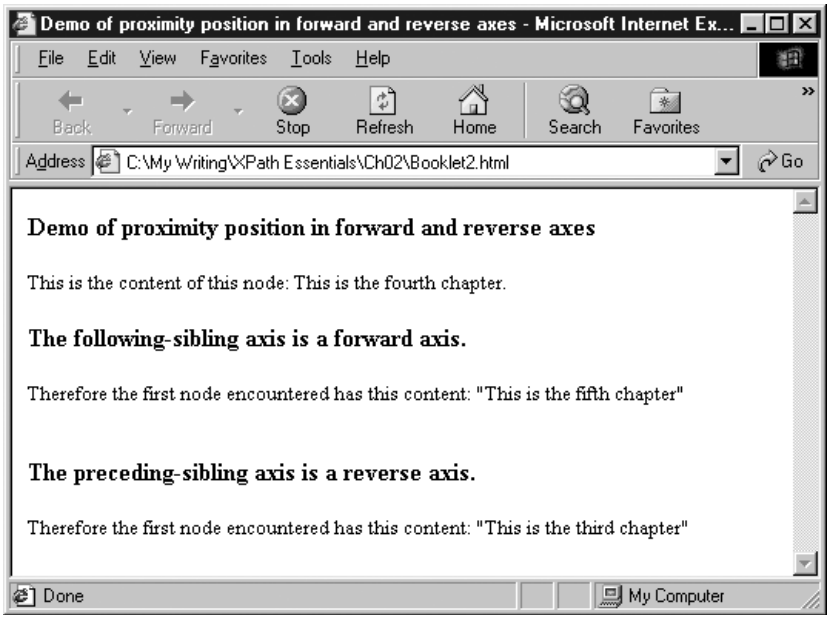

Demo of proximity position in forward and reverse axes - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Booklet2.html
Go
Demo of proximity position in forward and reverse axes
This is the content of this node: This is the fourth chapter.
The following-sibling axis is a forward axis.
Therefore the first node encountered has this content: "This is the fifth chapter"
The preceding-sibling axis is a reverse axis.
Therefore the first node encountered has this content: "This is the third chapter"
Done
My Computer

Figure 2.15 Demonstration of Proximity Position in Forward and Reverse Axes.

expect the content of the <Chapter> element immediately before the fourth <Chapter> element to be output. And this is what we find—the content of the third <Chapter> element is displayed.

Having explored the concepts and some examples of XPath location paths and their component parts—the axes, node tests, and predicates— let's move on and take a look at the core function library provided in the XML Path Language.

XPath Functions

This section will provide you with an introduction to XPath functions. Further details and examples of their use will be found in Chapter 5, "XPath Functions."

XPath functions return an object of one of four different types:

- Node set
- Number
- String
- Boolean

In this section we will examine the basics of how XPath functions are used.

Node Set Functions

XPath provides a number of functions that allow us to refine or filter a selection in a node set and then return another node set. XPath node set functions are often used in the predicates of location steps, as you have seen in a number of examples earlier in this chapter.

Let's take a brief look at each of the node set functions.

The count ( ) Function

The count ( ) function returns the number of nodes in an argument node set. Thus, with the example source document shown in Listing 2.30, we can use the count ( ) function to calculate how many <Chapter> elements the source document contains, as shown in Listing 2.31.

```
<?xml version='1.0'?>
<Chapters>
<Chapter order="first">The first chapter.</Chapter>
<Chapter order="second">The second chapter.</Chapter>
<Chapter order="third">The third chapter.</Chapter>
<Chapter order="fourth">The fourth chapter.</Chapter>
</Chapters>
```

Listing 2.30 The Chapters in a Book Expressed in XML (Chapters.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the XPath count() function</title>
</head>
<body>
<p>In the source document there are <xsl:value-of
select="count(Chapters/Chapter)"/> &lt;Chapter&gt; elements.</p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.31 XSLT Stylesheet to Transform Chapters.xml (Chapters.xsl).

Notice that the parameter passed to the count ( ) function is itself an XPath location path.

The id ( ) Function

The id ( ) function selects nodes based on their possessing a unique ID attribute.

To be designated as an ID attribute the attribute must be declared as being so in an accompanying Document Type Definition.

NOTE Version 1.0 of the XPath specification refers only to the Document Type Definition when referring to how an ID attribute is defined. It is likely that XPath 2.0 will incorporate a similar reference to XSD Schema (sometimes called simply "XML Schema"). For further information on XSD Schema see www.w3.org/TR/xmlschema-0/ and the links that that document contains.

Thus in Listing 2.32, we could use the isbn attribute as a unique identifier, since each published book has a unique ISBN.

```
<?xml version='1.0'?>
<ISBNNumbers>
<Book isbn="0-471-34402-8">Applied XML</Book>
<Book isbn="0-471-32753-0">XML Specification Guide</Book>
</ISBNNumbers>
```

Listing 2.32 A Short Catalog of Books in XML (ISBNNumbers0.xml).

However, if we wanted XML to be aware that we expect that the isbn attribute is to be unique, and more specifically is to be an ID attribute, we need to declare that to be so within a Document Type Definition, like this

```
<!ELEMENT ISBNNumbers (Book)*>
<!ELEMENT Book (#PCDATA)>
<!ATTLIST Book isbn ID #REQUIRED>
```

The third line defines the isbn attribute as being an ID attribute. With that in place we can use the XPath id ( ) function.

Thus, if we add that to the internal subset of the DTD, then our example file looks like Listing 2.33.

If we wanted to locate uniquely the book with the ISBN 0-471-32753-0, we could do so using the stylesheet in Listing 2.34.

```
<?xml version='1.0'?>
<!DOCTYPE ISBNNumbers [
<!ELEMENT ISBNNumbers (Book)*>
<!ELEMENT Book (#PCDATA)>
<!ATTLIST Book isbn ID #REQUIRED >
]>
<ISBNNumbers>
<Book isbn="ISBN0-471-34402-8">Applied XML</Book>
<Book isbn="ISBN0-471-32753-0">XML Specification Guide</Book>
</ISBNNumbers>
```

Listing 2.33 The Short Catalog of Books with an "Internal Subset" DTD Added (ISBNNumbers.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the XPath id() function</title>
</head>
<body>
<h3>Using the id() function</h3>
<p>Selected using the @isbn expression:
<xsl:value-of select="//Book[@isbn='ISBN0-471-32753-0']"/></p>
```

Listing 2.34 Selecting a Book by ID Attribute (ISBNNumbers.xsl).

```
<xsl:apply-templates select="id('ISBN0-471-32753-0')"/>
</body>
</html>
</xsl:template>

<xsl:template match="id('ISBN0-471-32753-0')">
<xsl:if test="id('ISBN0-471-32753-0')">
<p>The title of the book with isbn attribute of <xsl:value-of
select="@isbn"/> is <xsl:value-of select="."/>.</p>
</xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.34 (*Continued*)

The output we obtain on screen is shown in Figure 2.16.

There are several things which you must get right if you are to avoid a lot of frustration when using the id ( ) function:

- The attribute being referred to as an ID attribute must be so declared in a DTD.
- You cannot start the value of an ID attribute with a number.
- The value of an ID attribute may not contain a space character.

Also be aware that a number of XML books give an incorrect syntax for declaring an ID attribute. Additionally, at least two XML tools and one XSLT processor don't correctly identify the cause of a failed transformation.

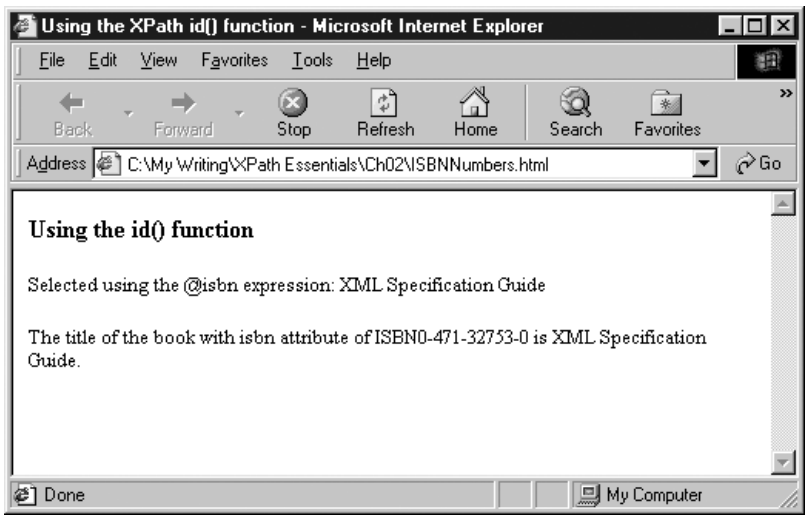

Using the XPath id() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\ISBNNumbers.html
Go
Using the id() function
Selected using the @isbn expression: XML Specification Guide
The title of the book with isbn attribute of ISBN0-471-32753-0 is XML Specification Guide.
Done
My Computer

Figure 2.16 Using the id ( ) function.

Since, as I demonstrated in the code, you could access the value of the isbn attribute more directly you could be forgiven for thinking that using the id ( ) function is sometimes more trouble than it is worth.

The last ( ) Function

The last ( ) function returns a number equal to the context size in the context where evaluation is taking place.

Thus, in Listing 2.35, we have a context size of 5 when the context node is any of the element nodes representing a <Section> element.

If we want to selectively display the content of only the fifth <Section> element, we can do so using the last ( ) function, as in the stylesheet in Listing 2.36.

Figure 2.17 illustrates the output when the stylesheet is applied to the source document.

The last ( ) function comes in particularly handy when we don't know, or don't wish to take the time to find out, what the context size is. Whatever the context size, the last ( ) function will return the node in the last position.

```
<?xml version='1.0'?>
<Chapter>
<Section>Placeholder text for the first section.</Section>
<Section>Placeholder text for the second section.</Section>
<Section>Placeholder text for the third section.</Section>
<Section>Placeholder text for the fourth section.</Section>
<Section>Placeholder text for the fifth section.</Section>
</Chapter>
```

Listing 2.35 The Sections in a Chapter Expressed in XML (Chapter.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the XPath last() function.</title>
</head>
<body>
<p>The XPath last() function returns an integer equal to the context
size.</p>
```

Listing 2.36 A Stylesheet Using the XPath last ( ) Function (Chapter.xsl).

```
<p>Here is the content of the last node in the context:</p>
<xsl:apply-templates select="Chapter/Section[last()]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Section">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.36 (*Continued*)

The local-name ( ) Function

The XPath specification states about the local-name ( ) function, "The local-name function returns the local part of the expanded-name of the node in the argument node-set that is first in document order."

NOTE **The local part is the part of a QName after the colon. So in the element <xmml:Book>, the string "Book" is the local part.**

To demonstrate the use of the local-name ( ) function, we need a source document that has elements which have a namespace prefix and a local part (see Listing 2.37).

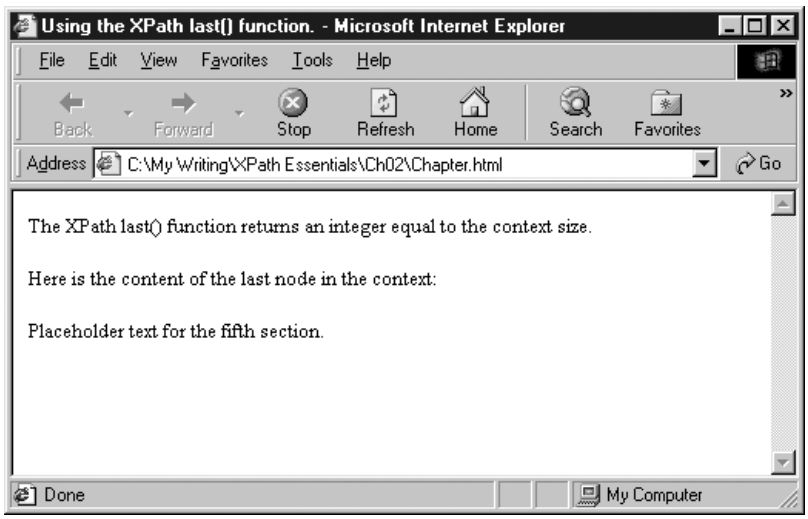

Using the XPath last() function. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Chapter.html
Go
The XPath last() function returns an integer equal to the context size.
Here is the content of the last node in the context:
Placeholder text for the fifth section.
Done
My Computer

Figure 2.17 Selecting the Last Node Using the last ( ) Function.

```
<?xml version='1.0'?>
<XMML:Book xmlns:XMML="http://www.XMML.com/BookSchemas/">
<XMML:Chapters>
<XMML:Chapter>This is the first chapter</XMML:Chapter>
<XMML:Chapter>This is a second chapter</XMML:Chapter>
<XMML:Chapter>This is a third chapter</XMML:Chapter>
</XMML:Chapters>
</XMML:Book>
```

Listing 2.37 Structure of a Book in XML Using a Namespace (XMMLBook.xml).

We can use the stylesheet in Listing 2.38 to output the value of the local part of the QName in the element <XMML:Chapter>.

In Figure 2.18 we see that the desired information is output on screen.

Remember that it is only the local part of the first name in a node set that is output. So if we modified our source document (see Listing 2.39) and modified the <xsl:value-of> element to

```
<xsl:value-of
select="local-name(XMML:Book/XMML:*)"/>
```

the answer, which is output, is "Introduction," which is the local part of the <XMML:Introduction> element—the first element in document order.

TEAMFLY

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:XMML="http://www.XMML.com/BookSchemas/">

<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<p>The local part of the &lt;XMML:Chapter&gt; element is <xsl:value-of
select="local-name(XMML:Book/XMML:Chapters/XMML:Chapter)"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.38 A Stylesheet Demonstrating the local-name ( ) Function (XMMLBook.xsl).

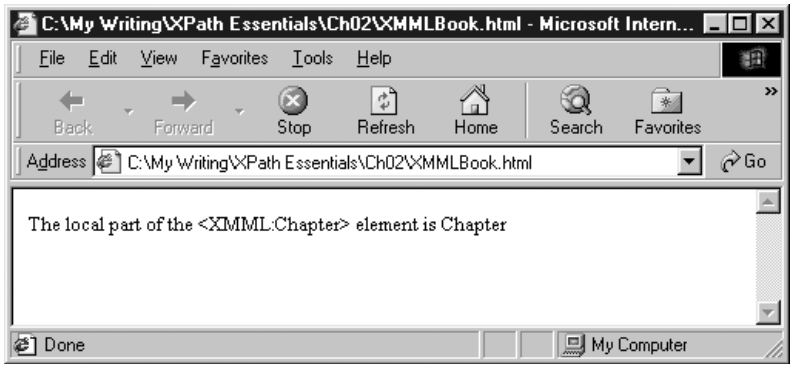

C:\My Writing\XPath Essentials\Ch02\XMMLBook.html - Microsoft Intern...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\XMMLBook.html
Go
The local part of the <XMML:Chapter> element is Chapter
Done
My Computer

Figure 2.18 Demonstration of the XPath local-name ( ) Function.

```
<?xml version='1.0'?>
<XMML:Book xmlns:XMML="http://www.XMML.com/BookSchemas/">
<XMML:Introduction>Some introductory material</XMML:Introduction>
<XMML:Chapter>This is the first chapter</XMML:Chapter>
<XMML:Chapter>This is a second chapter</XMML:Chapter>
<XMML:Chapter>This is a third chapter</XMML:Chapter>
<XMML:Appendix>Some material of an appendix-like nature.</XMML:Appendix>
</XMML:Book>
```

Listing 2.39 A Modified Version of the Book Structure (XMMLBook2.xml).

The name ( ) Function

The XPath specification says this about the name ( ) function: "The name function returns a string containing a QName representing the expanded-name of the node in the argument node-set that is first in document order."

Let's use the Scalable Vector Graphics language (SVG) for our example source document (see Listing 2.40).

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
     "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg:svg xmlns:svg="http://www.w3.org/2000/svg" svg:width="500"
svg:height="500">
<svg:rect x="50" y="100" width="100" height="20"
style="stroke:#000099; stroke-width:2; fill:#FFCCFF"/>

</svg:svg>
```

Listing 2.40 An SVG Source Document (Rect.svg).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:svg="http://www.w3.org/2000/svg">

<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<p>The name() function applied to the &lt;svg:svg&gt; element returns:
  <xsl:value-of select="name(svg:svg)"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.41 A Stylesheet Demonstrating the name ( ) Function (Rect.xsl).

The stylesheet in Listing 2.41 makes use of the name ( ) function and applies it to the <svg:svg> element.

As you can see in Figure 2.19, the name ( ) function returns the QName of the element.

As used in the example, the name ( ) function is pretty uninspiring but it could be useful when producing error messages to assist in debugging a stylesheet, for example. If the name ( ) function is used, then it is possible to see where the processing has reached, when an error occurred.

NOTE **To carry out the above transformation, you may need to be online since, for example, Instant Saxon has no internal knowledge of the SVG DTD, and therefore you need to be online for Instant Saxon to be able to access the DTD directly. If you are offline while attempting this transformation, you will likely get an error message something like "Error java.net.UnknownHost Exception: www.w3.org: www.w3.org."**

The namespace-uri ( ) Function

The namespace-uri ( ) function returns the namespace URI of the node(s) in the argument node set.

Thus if we had the code

```
namespace-uri(W3C:SomeElement)
```

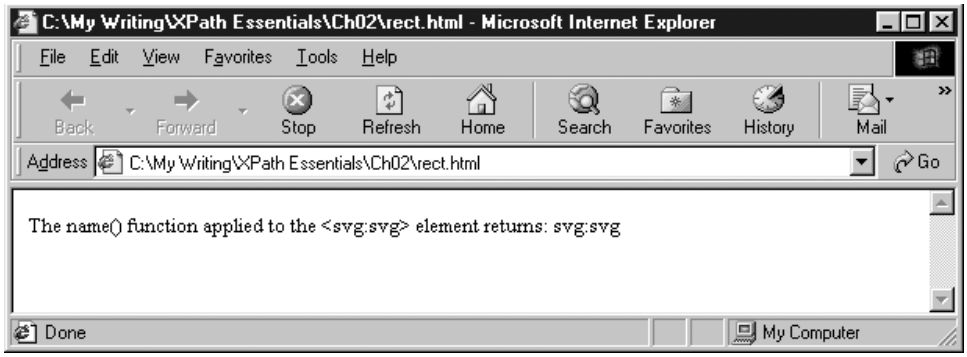

C:\My Writing\XPath Essentials\Ch02\rect.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch02\rect.html
Go
The name() function applied to the <svg:svg> element returns: svg:svg
Done
My Computer

Figure 2.19 Demonstration of the name ( ) Function.

where <W3C:SomeElement> had been declared like this

```
<W3C:SomeElement xmlns:W3C="http:www.w3.org/SomeFictionalNamespace/">
```

then the namespace-uri ( ) function would return the URI of "http://www.w3.org/SomeFictionalNamespace/".

The position ( ) Function

The XPath position ( ) function returns a number that reflects the context position of the context node (see Listing 2.42).

We can use the position ( ) function to select only certain nodes or, as in the stylesheet in Listing 2.43, we can output explicitly the value returned by the position ( ) function.

The position ( ) function also allows us to use the ">" (greater than) or "<" (less than) operators. Thus if we wanted to output only those nodes where the position was greater than two, we could modify the <xsl:apply-templates> element as follows:

```
<xsl:apply-templates select="RaceResult/Place[position()>2]"/>
```

The output of the modified stylesheet, RaceResult2.xsl, is shown in Figure 2.20.

If you look closely at the output shown in Figure 2.20, you will see that a new meaning of "position" is apparent. When the <xsl:apply-templates> element, as modified, is

```
<?xml version='1.0'?>
<RaceResult>
<Place order="first">Michael Schumacher</Place>
<Place order="second">Ralph Schumacher</Place>
<Place order="third">Rubens Barichello</Place>
<Place order="fourth">David Coulthard</Place>
</RaceResult>
```

Listing 2.42 The Result of a Formula 1 Race Expressed in XML (RaceResult.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<xsl:apply-templates select="RaceResult/Place"/>
</body>
</html>
</xsl:template>

<xsl:template match="Place">
<p>The element node in <xsl:value-of select="@order"/> place is in
  position <xsl:value-of select="position()"/>.</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.43 A Stylesheet Demonstrating the Use of the position ( ) Function (RaceResult.xsl).

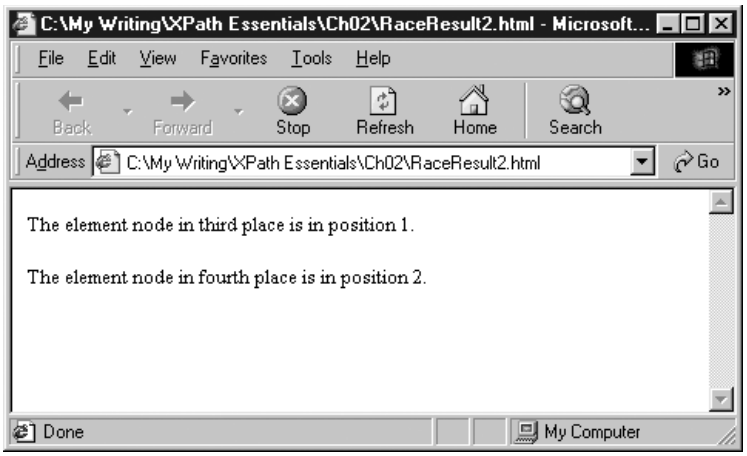

C:\My Writing\XPath Essentials\Ch02\RaceResult2.html - Microsoft...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\RaceResult2.html
Go
The element node in third place is in position 1.
The element node in fourth place is in position 2.
Done
My Computer

Figure 2.20 Using the XPath position ( ) Function.

applied then a new node set containing only two nodes is selected. Thus the element node for the <Place> element whose order attribute has the value of "third" is in position 1 in the new node set, which has a context size of 2. Similarly the element node for the last <Place> element is in position 2, as demonstrated by the output shown.

Number Functions

XPath provides some simple functions to manipulate numbers. In this section I will briefly describe each of them with a simple example illustrating their use.

The ceiling ( ) Function

The ceiling ( ) function returns the smallest integer, which is greater than the argument for the ceiling ( ) function. It is essentially a "round up" function.

With the source document shown in Listing 2.44, we can use the ceiling ( ) function to round up each of the daily temperatures prior to display, as in the stylesheet in Listing 2.45.

```
<?xml version='1.0'?>
<DailyTemperature Units="Fahrenheit">
<Monday>61.3</Monday>
<Tuesday>68.1</Tuesday>
<Wednesday>64.9</Wednesday>
<Thursday>71.0</Thursday>
<Friday>65.4</Friday>
</DailyTemperature>
```

Listing 2.44 Temperatures on a Series of Weekdays (DailyTemperature2.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Daily Temperatures - rounded up using the ceiling()
function</title>
</head>
<body>
<h3>Daily Temperatures - rounded up, using ceiling() function</h3>
<xsl:apply-templates select="DailyTemperature/*"/>
</body>
</html>
</xsl:template>

<xsl:template match="Monday | Tuesday | Wednesday | Thursday | Friday">
<p><xsl:value-of select="ceiling(.)"/> - <xsl:value-of select="name()"
  /></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.45 A Stylesheet Demonstrating the ceiling ( ) Function (DailyTemperature2.xsl).

The floor ( ) Function

The floor ( ) function returns the largest integer that is not greater than the argument for the floor ( ) function. It is essentially a "round down" function.

With the source document shown in Listing 2.46, we can use the floor ( ) function in the stylesheet in Listing 2.47 to round down the daily temperature and display it for each day.

```
<?xml version='1.0'?>
<DailyTemperature Units="Fahrenheit">
<Monday>61.3</Monday>
<Tuesday>68.1</Tuesday>
<Wednesday>64.9</Wednesday>
<Thursday>71.0</Thursday>
<Friday>65.4</Friday>
</DailyTemperature>
```

Listing 2.46 Temperatures on a Series of Weekdays (DailyTemperature.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Daily Temperatures - using the floor() function</title>
</head>
<body>
<h3>Daily Temperatures - rounded down, using floor() function</h3>
<xsl:apply-templates select="DailyTemperature/*"/>
</body>
</html>
</xsl:template>

<xsl:template match="Monday | Tuesday | Wednesday | Thursday | Friday">
<p><xsl:value-of select="floor(.)"/> - <xsl:value-of
select="name()"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.47 A Stylesheet Demonstrating the floor ( ) Function (DailyTemperature.xsl).

The number ( ) Function

The number ( ) function converts its argument to a number.

If we have the source document shown in Listing 2.48, it will allow us to see, using Listing 2.49, what the number ( ) function does when presented with string, node set, and Boolean arguments.

Figure 2.21 shows the results of the number ( ) function applied to those three types of arguments.

```
<?xml version='1.0'?>
<OddDocument>
<SomeText>A string.</SomeText>
<ANumber>34</ANumber>
</OddDocument>
```

Listing 2.48 A Source Document to Demonstrate the number ( ) Function (OddDocument.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the number() function</title>
</head>
<body>
<h3>Using the number() function</h3>
<p> Converting a string value of "<xsl:value-of
select="string(OddDocument/SomeText)"/>":
<xsl:value-of select="number(string(OddDocument/SomeText))"/></p>
<p> Converting a node set:
<xsl:value-of select="number(OddDocument/SomeText)"/></p>
<p> Converting a boolean: <xsl:value-of select="number(false())"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.49 A Stylesheet Demonstrating the number ( ) Function (OddDocument.xsl).

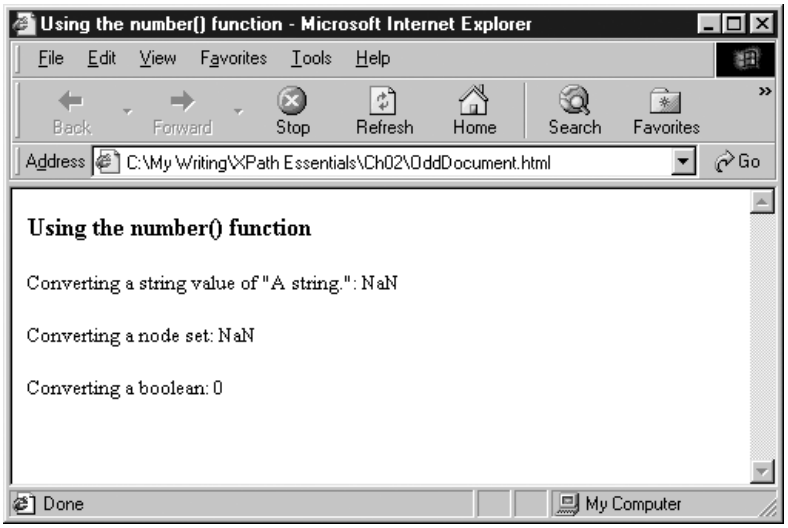

Using the number() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\OddDocument.html
Go
Using the number() function
Converting a string value of "A string.": NaN
Converting a node set: NaN
Converting a boolean: 0
Done
My Computer

Figure 2.21 Demonstration of the number ( ) Function.

The false ( ) function always returns the Boolean value of "false." When converted to a number this gives the value of zero.

The round ( ) Function

The round ( ) function rounds a real number to the nearest integer.

If we have the example source document shown in Listing 2.50, we can use the stylesheet in Listing 2.51 to output the monthly sales figures, rounding each real number to an integer.

The output is shown in Figure 2.22.

Notice that the value of 17.49 is rounded down and the value of 18.51 is rounded up, as you would probably expect. However, the value of 21.50 is rounded up. The XPath round ( ) function rounds toward positive infinity if there is value equally distant from the next lower and next higher integer.

```
<?xml version='1.0'?>
<Q1Sales Year="2003">
<January>17.49</January>
<February>18.51</February>
<March>21.50</March>
</Q1Sales>
```

Listing 2.50 Quarter 1 Sales for 2003 Expressed in XML (Q1Sales.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Q1 Sales for XMML.com: <xsl:value-of
select="Q1Sales/@Year"/></title>
</head>
<body>
<h3>Quarter 1 Sales for <xsl:value-of select="Q1Sales/@Year"/> by
  month</h3>
<p>Sales rounded to nearest whole number ($000's)
<xsl:apply-templates select="Q1Sales/*"/></p>
</body>
</html>
</xsl:template>

<xsl:template match="January | February | March">
<p>Sales for <xsl:value-of select="name()"/> were <xsl:value-of
  select="round(.)"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.51 A Stylesheet to Demonstrate the round ( ) Function (Q1Sales.xsl).

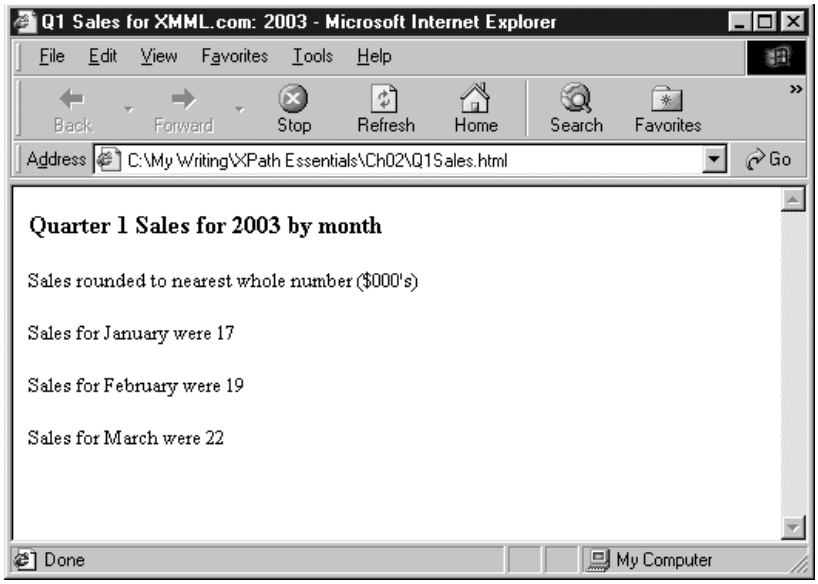

Q1 Sales for XMML.com: 2003 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Q1Sales.html
Go
Quarter 1 Sales for 2003 by month
Sales rounded to nearest whole number ($000's)
Sales for January were 17
Sales for February were 19
Sales for March were 22
Done
My Computer

Figure 2.22 Demonstration of the round ( ) Function.

The sum ( ) Function

The sum ( ) function returns the sum of adding the string-value of each node in a node set, after conversion to a number.

If we again use the Q1Sales figures as source document (see the round ( ) function), we can apply a stylesheet like Listing 2.52 to sum the sales figures for the quarter.

The sum ( ) function carries out the string to number conversion automatically. There is no need to use the number ( ) function, in this context.

String Functions

XPath provides some simple functions to manipulate strings. In this section I will briefly describe each of them with a simple example illustrating their use.

The concat ( ) Function

The concat ( ) function concatenates the arguments passed to it.

Thus if the concat ( ) function was applied to the source document shown in Listing 2.53, we can use the concat ( ) function as in the stylesheet in Listing 2.54 to combine the two phrases.

TEAMFLY

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Q1 Sales for XMML.com: <xsl:value-of
select="Q1Sales/@Year"/></title>
</head>
<body>
<h3>Quarter 1 Sales for <xsl:value-of select="Q1Sales/@Year"/> by
  month</h3>
<p>Sales rounded to nearest whole number ($000's)</p>
<p>The quarterly sales totalled ($ 000's): $<xsl:value-of
  select="sum(Q1Sales/*)"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.52 A Stylesheet to Demonstrate the sum ( ) Function (Q1Sales2.xsl).

```
<?xml version='1.0'?>
<NurseryRhyme>
<Phrase>Mary had a</Phrase>
<Phrase>little lamb.</Phrase>
</NurseryRhyme>
```

Listing 2.53 A Traditional Nursery Rhyme (NurseryRhyme.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<p>
The start of the nursery rhyme is:
<xsl:value-of select="concat(string(NurseryRhyme/Phrase[position()=1]),'
', string(NurseryRhyme/Phrase[position()=2]))"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.54 A Stylesheet to Demonstrate the concat ( ) Function (NurseryRhyme.xsl).

The value of the select attribute of the <xsl:value-of> element is fairly complex, so let's break it down into its component parts.

```
select="concat(string(NurseryRhyme/Phrase[position()=1]),' ',string
  (NurseryRhyme/Phrase[position()=2]))"
```

It takes the basic form of three string arguments, separated by commas

```
concat(string1, string2, string3)
```

The first string argument is

```
string(NurseryRhyme/Phrase[position()=1])
```

which means the string ( ) function applied to the node set selected by the location path `NurseryRhyme/Phrase[position()=1]`, which has the string value of "Mary had a."

The second string argument is

```
' '
```

which is simply a single quote followed by a space character followed by a single quote.

The third string argument is similar to the first and is

```
string(NurseryRhyme/Phrase[position()=2]
```

which applies the string ( ) function to the node set (a single node) selected by the `location`

```
path NurseryRhyme/Phrase[position()=2].
```

Thus, effectively the concat ( ) function looks like this

```
concat("Mary had a", ' ', "little lamb")
```

This is much easier, perhaps, to understand than when you first saw the value of the select attribute.

The contains ( ) Function

The contains ( ) function takes two string arguments. It evaluates whether the second string is contained in the first. If it does, then it returns the Boolean value of "true." Otherwise it returns the Boolean value of "false."

With the source document shown in Listing 2.55, we can test whether or not a node contains the string "Mary" and carry out processing of a template based on the result of that test, as in the stylesheet in Listing 2.56.

Notice how the contains ( ) function is used in the test attribute of the <xsl:if> element to determine whether the content of the <xsl:template> element is instantiated or not.

The output of the stylesheet can be seen in Figure 2.23.

```
<?xml version='1.0'?>
<Rhymes>
<Rhyme>Mary had a little lamb</Rhyme>
<Rhyme>Humpty, Dumpty sat on a wall</Rhyme>
<Rhyme>Mary, Mary quite contrary</Rhyme>
</Rhymes>
```

Listing 2.55 A Short Catalog of Nursery Rhymes (Rhymes.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the contains() function</title>
</head>
<body>
<h3>Using the XPath contains() function</h3>
<xsl:apply-templates select="Rhymes/Rhyme"/>
</body>
</html>
</xsl:template>

<xsl:template match="Rhyme">
<xsl:if test="contains(., 'Mary')">
<p>The rhyme "<xsl:value-of select="."/>" contains the word "Mary"</p>
</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.56 A Stylesheet to Demonstrate the contains ( ) Function (Rhymes.xsl).

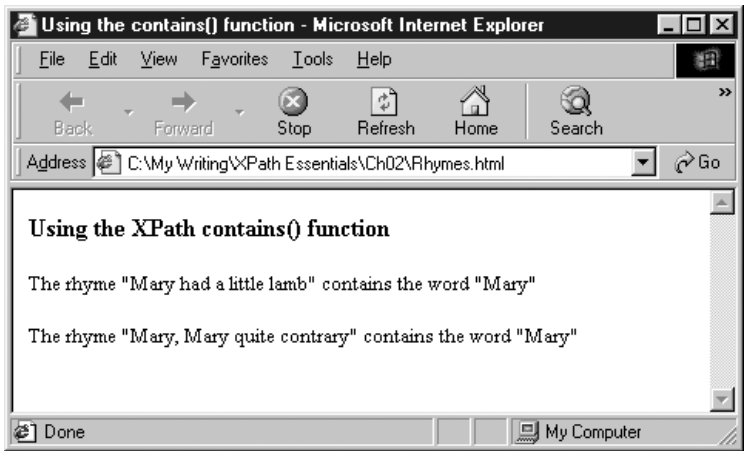

Using the contains() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\Rhymes.html
Go
Using the XPath contains() function
The rhyme "Mary had a little lamb" contains the word "Mary"
The rhyme "Mary, Mary quite contrary" contains the word "Mary"
Done
My Computer

Figure 2.23 A Demonstration of the contains ( ) Function.

The normalize-space ( ) Function

The normalize-space ( ) function strips leading or excess whitespace characters from a string.

```
<?xml version='1.0'?>
<OddSpacing>
<SomeText>            This has leading spaces and one       big gap
  caused by excess whitespace.</SomeText>
<SomeText>This doesn't          have leading space        but   does
  have sequences    of   space characters   .</SomeText>
</OddSpacing>
```

Listing 2.57 A Source Document with Excess Whitespace Characters (OddSpacing.xml).

Let's create a fairly bizarre source document with some leading space characters and some spare ones within the text (see Listing 2.57).

We can use the normalize-space ( ) function, as in the stylesheet shown in Listing 2.58, to remove the leading spaces and to convert sequences of spaces to single space characters.

Of course, since HTML browsers automatically strip out excess whitespace, we need to look at the generated HTML (see Listing 2.59) to see the effect.

The starts-with ( ) Function

The starts-with ( ) function takes two string arguments. It evaluates whether the first string argument starts with the second string argument.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the normalize-space() function</title>
</head>
<body>
<h3>Using normalize-space() - the "before" and "after"</h3>
<xsl:apply-templates select="OddSpacing/SomeText"/>
</body>
</html>
</xsl:template>

<xsl:template match="SomeText">
<p>Before:<xsl:value-of select="."/></p>
<p> After:<xsl:value-of select="normalize-space(.)"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.58 A Stylesheet to Demonstrate the normalize-space ( ) Function (OddSpacing.xsl).

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Using the normalize-space() function</title>
</head>
<body>
<h3>Using normalize-space() - the "before" and "after"</h3>
<p>Before:     This has leading spaces and one       big gap caused by
  excess whitespace.</p>
<p> After:This has leading spaces and one big gap caused by excess
  whitespace.</p>
<p>Before:This doesn't          have leading space        but   does
  have sequences    of   space characters   .</p>
<p> After:This doesn't have leading space but does have sequences of
  space characters .</p>
</body>
</html>
```

Listing 2.59 An HTML Document Output after Using the normalize-space ( ) Function (OddSpacing.html).

Thus if we again used the source document shown in Listing 2.60, we can use the starts-with ( ) function to determine whether the first <Phrase> element starts with the string "Mar" by using the stylesheet in Listing 2.61.

```
<?xml version='1.0'?>
<NurseryRhyme>
<Phrase>Mary had a</Phrase>
<Phrase>little lamb.</Phrase>
</NurseryRhyme>
```

Listing 2.60 A Brief Nursery Rhyme (NurseryRhyme.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
```

Listing 2.61 A Stylesheet to Demonstrate the starts-with ( ) Function (NurseryRhyme2.xsl). (*continues*)

```
<title></title>
</head>
<body>
<p>
Does the first phrase of the nursery rhyme start with "Mar"? -
<xsl:value-of select="starts-
with(string(NurseryRhyme/Phrase[position()=1]),'Mar')"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.61 (*Continued*)

The first phrase of the nursery rhyme does start with the string "Mar" so the answer to our question is "true" (see Figure 2.24).

The string ( ) Function

The string ( ) function converts its argument to a string.

For example, if we apply the stylesheet shown in Listing 2.63 to the source document shown in Listing 2.62, then we output the string value of the first node in the node set of element nodes that represent the <Brand> element.

It is straightforward to combine the string ( ) function with, say, the last ( ) function in a predicate as in

```
<xsl:value-of select="string(CameraBrands/Brand[last()])"/>
```

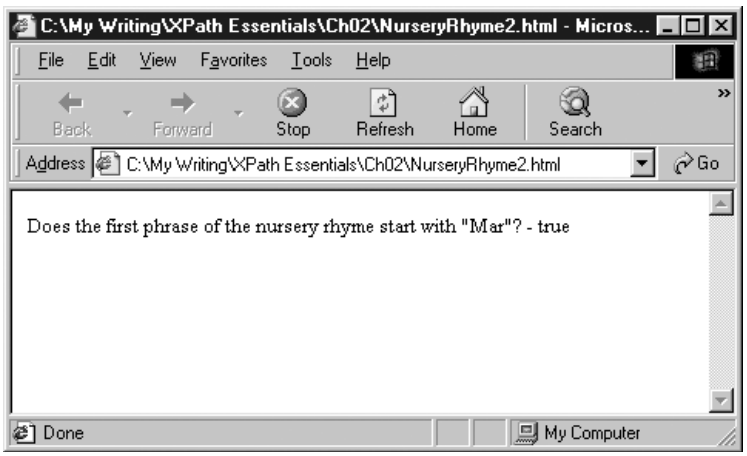

C:\My Writing\XPath Essentials\Ch02\NurseryRhyme2.html - Micros...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\NurseryRhyme2.html
Go
Does the first phrase of the nursery rhyme start with "Mar"? - true
Done
My Computer

Figure 2.24 A Demonstration of the starts-with ( ) Function.

```
<?xml version='1.0'?>
<CameraBrands>
<Brand>Canon</Brand>
<Brand>Nikon</Brand>
<Brand>Pentax</Brand>
<Brand>Praktica</Brand>
</CameraBrands>
```

Listing 2.62 A Brief Catalog of Camera Brands (CameraBrands.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>
<p>The first &lt;Brand&gt; node when converted by the string() function
  has
the value of <xsl:value-of select="string(CameraBrands/Brand)"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 2.63 A Stylesheet to Demonstrate the string ( ) Function (CameraBrands.xsl).

that produces the result of applying the string ( ) function to the last of the <Brand> elements. The output is shown in Figure 2.25.

In the above examples I have used the string ( ) function with element nodes. However, it may also be used with other node types. The string ( ) function is described in more detail in Chapter 5, "XPath Functions."

The string-length ( ) Function

The string-length ( ) function returns the number of characters in the string, which is its argument.

If we have the source document shown in Listing 2.64, we can evaluate the length of the string in each <String> element using the string-length() function, as in the stylesheet in Listing 2.65.

The output from the stylesheet is shown in Figure 2.26.

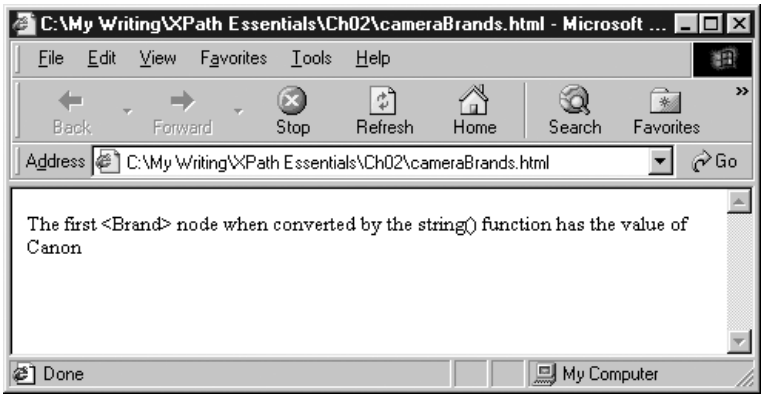

C:\My Writing\XPath Essentials\Ch02\cameraBrands.html - Microsoft ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\cameraBrands.html
Go
The first <Brand> node when converted by the string() function has the value of
Canon
Done
My Computer

Figure 2.25 Demonstration of the string ( ) Function.

```
<?xml version='1.0'?>
<Lengths>
<String>Short</String>
<String>A medium length.</String>
<String>This is considerably longer than the preceding two
strings.</String>
</Lengths>
```

Listing 2.64 A Collection of Strings of Differing Lengths (Lengths.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the string-length() function</title>
</head>
<body>
<h3>Using the string-length() function</h3>
<xsl:apply-templates select="Lengths/String"/>
</body>
</html>
</xsl:template>

<xsl:template match="String">
<p>The string "<xsl:value-of select="."/>" in position <xsl:value-of
  select="position()"/>
```

Listing 2.65 A Stylesheet Demonstrating the string-length ( ) Function (Lengths.xsl).

```
is <xsl:value-of select="string-length(.)"/> characters long.</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.65 (*Continued*)

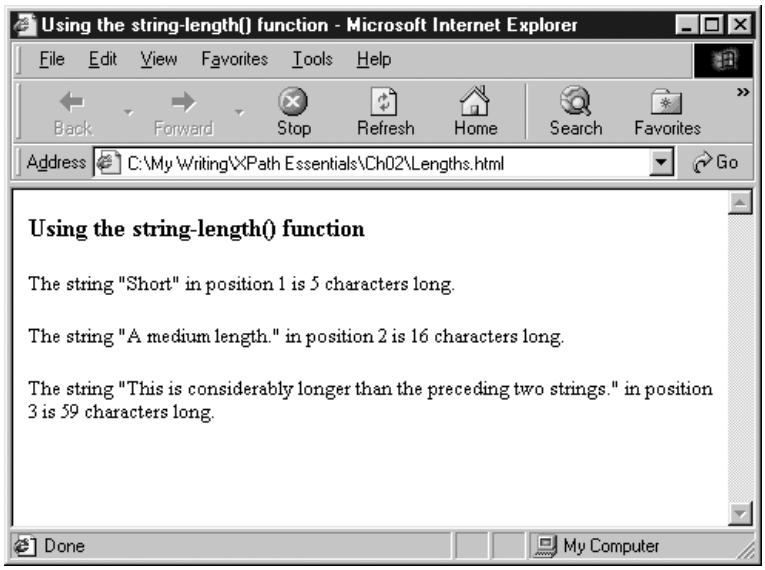

Using the string-length() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Lengths.html
Go
Using the string-length() function
The string "Short" in position 1 is 5 characters long.
The string "A medium length." in position 2 is 16 characters long.
The string "This is considerably longer than the preceding two strings." in position 3 is 59 characters long.
Done
My Computer

Figure 2.26 Demonstration of the string-length ( ) Function.

The substring ( ) Function

The substring ( ) function returns the substring of the first string argument, starting at the character specified by its second string argument and of length specified by the third string argument.

With the source document shown in Listing 2.66, we can apply the substring ( ) function using the stylesheet in Listing 2.67.

The output is shown in Figure 2.27

```
<?xml version='1.0'?>
<Strings>
<String>This is a string.</String>
<String>This too is a string but not the same as the first.</String>
</Strings>
```

Listing 2.66 Strings to Use to Explore the substring ( ) Function (Strings.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the substring() function</title>
</head>
<body>
<h3>Using the substring() function.</h3>
<xsl:apply-templates select="Strings/String"/>
</body>
</html>
</xsl:template>

<xsl:template match="String">
<p>The substring of "<xsl:value-of select="."/>" beginning at character
  4 and of length 10 is
"<xsl:value-of select="substring(., 4, 10)"/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.67 A Stylesheet to Demonstrate the substring ( ) Function (Strings.xsl).

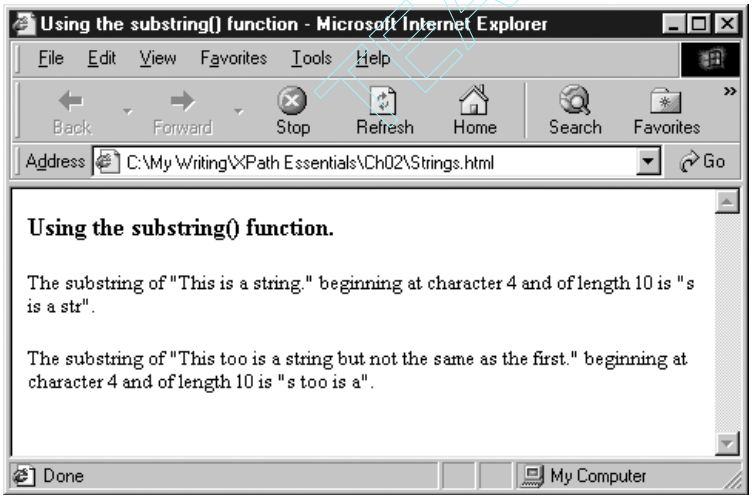

Using the substring() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch02\Strings.html
Go
Using the substring() function.
The substring of "This is a string." beginning at character 4 and of length 10 is "s is a str".
The substring of "This too is a string but not the same as the first." beginning at character 4 and of length 10 is "s too is a".
Done
My Computer

Figure 2.27 A Demonstration of the substring ( ) Function.

The substring-after ( ) Function

The substring-after ( ) function returns the substring that occurs in its first string argument after the first occurrence of the string in its second string argument. Thus, with the

source document shown in Listing 2.68, we can apply the substring-after ( ) function to return the string that occurs after the string "very" as in Listing 2.69.

The output of the stylesheet is shown in Figure 2.28.

The substring-before ( ) Function

The substring-before ( ) function returns the substring that occurs in its first string argument before the first occurrence of the string in its second string argument.

If we use the same source document as for the substring-after ( ) function and apply the stylesheet in Listing 2.70, we can see how the substring-before ( ) function works.

The output is seen in Figure 2.29.

```
<?xml version='1.0'?>
<MoreStrings>
<String>Mary had a very little lamb.</String>
<String>Dry ice is very cold.</String>
</MoreStrings>
```

Listing 2.68 Some Simple Strings (MoreStrings.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the substring-after() function</title>
</head>
<body>
<h3>Using the substring-after() function</h3>
<xsl:apply-templates select="MoreStrings/String"/>
</body>
</html>
</xsl:template>

<xsl:template match="String">
<p>The substring after the first occurrence of "very" in "<xsl:value-of
  select="."/>"
is "<xsl:value-of select="substring-after(., 'very')"/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.69 A Stylesheet to Demonstrate the substring-after ( ) Function (MoreStrings.xsl).

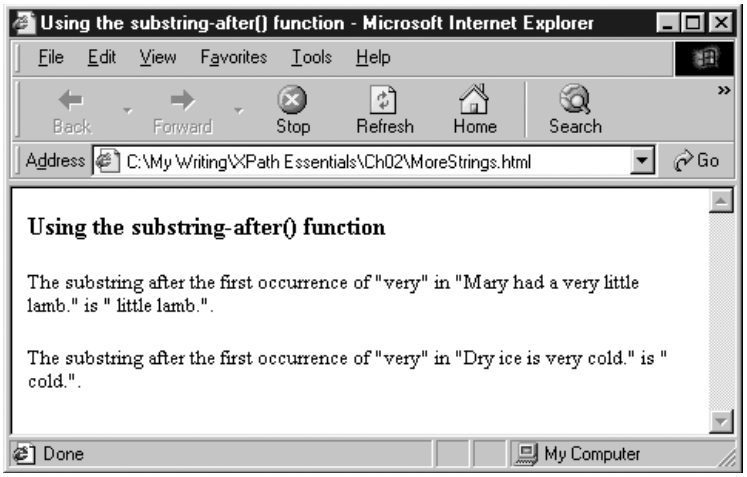

Using the substring-after() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\MoreStrings.html
Go
Using the substring-after() function
The substring after the first occurrence of "very" in "Mary had a very little lamb." is " little lamb.".
The substring after the first occurrence of "very" in "Dry ice is very cold." is " cold.".
Done
My Computer

Figure 2.28 A Demonstration of the substring-after ( ) Function.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the substring-before() function</title>
</head>
<body>
<h3>Using the substring-before() function</h3>
<xsl:apply-templates select="MoreStrings/String"/>
</body>
</html>
</xsl:template>

<xsl:template match="String">
<p>The substring before the first occurrence of "very" in "<xsl:value-of
  select="."/>"
is "<xsl:value-of select="substring-before(., 'very')"/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.70 A Stylesheet to Demonstrate the substring-before ( ) Function (MoreStrings2.xsl).

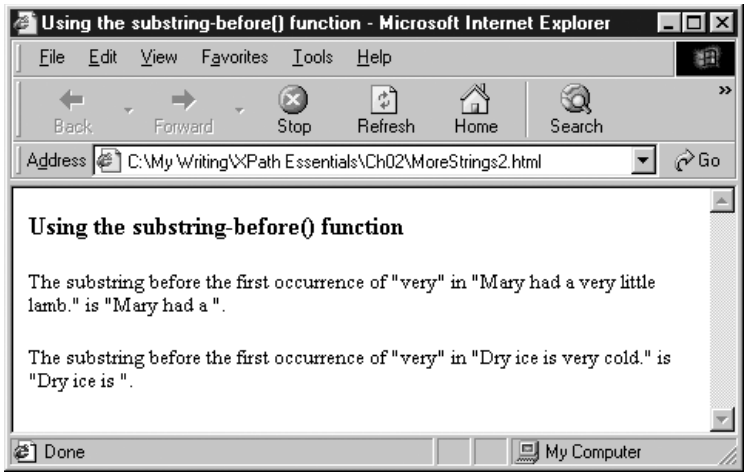

Using the substring-before() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\MoreStrings2.html
Go
Using the substring-before() function
The substring before the first occurrence of "very" in "Mary had a very little lamb." is "Mary had a ".
The substring before the first occurrence of "very" in "Dry ice is very cold." is "Dry ice is ".
Done
My Computer

Figure 2.29 A Demonstration of the substring-before ( ) Function.

The translate ( ) Function

The XPath specification says of the translate ( ) function, "The translate function returns the first argument string with occurrences of characters in the second argument string replaced by the character at the corresponding position in the third argument string."

Thus, with the source document shown in Listing 2.71, we can change lowercase letters to uppercase by using the stylesheet in Listing 2.72.

The output of the translate ( ) function is shown in Figure 2.30.

```
<?xml version='1.0'?>
<SomeStrings><String>ALL UPPER CASE</String>
<String>not upper case yet</String>
<String>mIxeD Up or DoWn cAsE</String>
</SomeStrings>
```

Listing 2.71 A Selection of Strings of Different Cases (SomeStrings.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
```

Listing 2.72 A Stylesheet to Demonstrate the translate ( ) Function (SomeStrings.xsl).

(*continues*)

```
<html>
<head>
<title>Using the translate() function</title>
</head>
<body>
<h3>Using the translate() function</h3>
<xsl:apply-templates select="SomeStrings/String"/>
</body>
</html>
</xsl:template>

<xsl:template match="String">
<p>Applying the translate() function specified to the string
  "<xsl:value-of select="."/>" produces the string
"<xsl:value-of select="translate(., 'abcdefghijklmnopqrstuvwxyz',
  'ABCDEFGHIJKLMNOPQRSTUVWXYZ')"/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 2.72 (*Continued*)

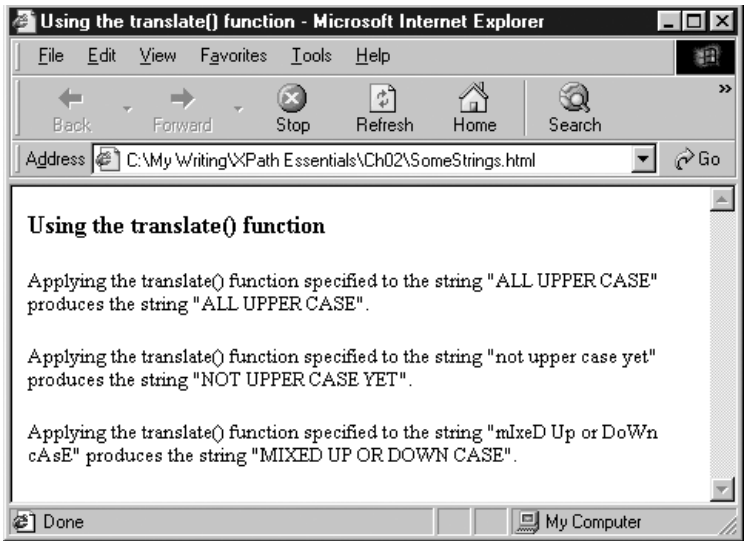

Using the translate() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch02\SomeStrings.html
Go
Using the translate() function
Applying the translate() function specified to the string "ALL UPPER CASE" produces the string "ALL UPPER CASE".
Applying the translate() function specified to the string "not upper case yet" produces the string "NOT UPPER CASE YET".
Applying the translate() function specified to the string "mIxeD Up or DoWn cAsE" produces the string "MIXED UP OR DOWN CASE".
Done
My Computer

Figure 2.30 A Demonstration of the translate ( ) Function. (02XPA30.tif)

Boolean Functions

XPath 1.0 provides five Boolean functions.

The boolean( ) function. The boolean ( ) function converts its argument to a Boolean value.

The false( ) function. The false ( ) function returns the Boolean value of "false" whatever its argument.

The lang( ) function. The lang function returns true or false, depending on whether the language of the context node as specified by `xml:lang` attributes is the same as or is a sublanguage of the language specified by the argument string.

The not( ) function. The not ( ) function returns false if its argument is true, and returns true otherwise.

The true( ) function. The true ( ) function returns the Boolean value of "true."

PATTERNS

In this and later chapters I use the term *pattern*. It is worth taking a little time to explain what a "pattern" is.

At its simplest, a pattern is a description of which nodes a template rule applies to. Or, to express it another way, a pattern is a test that can be applied to nodes to see if they match.

The main use of an XSLT pattern is in the match attribute of the <xsl:template> element. However, they can also be used in the match attribute of the <xsl:key> element and in the count and from attributes of the <xsl:number> element.

Looking Ahead

In this chapter I provided an overview of many aspects of XPath and showed you many examples of how XPath works. In the following chapters I will examine in more detail the concepts and use of the XPath functionality we have looked at in this chapter. Let's move on to Chapter 3 and take a closer look at the XPath data model.

XPath Data Model

I indicated in Chapter 2 that XPath operates on an abstract, logical model of an XML document. In this chapter we will look at the XPath data model in greater detail and compare it to the data models of the Document Object Model (DOM) and the XML Information Set.

The W3C has produced three different data models since 1998 that model the content of an XML document in different ways and which don't, in their present forms, map readily from one to another. Of course there were good reasons for each of those models, within the context in which they were developed, but as we move toward a more fluid cross-technology use of XML, the presence of three different data models isn't ideal.

The primary emphasis of this chapter is to describe the XPath data model, but the Document Object Model and XML Information Set (often referred to as the *infoset*), will also be discussed.

The final part of the chapter is dedicated to the XML Information Set, which is the third data model for XML produced by the W3C. It is clear from recent W3C documents that the XML Information Set will be the most important model in W3C's strategy for XML, in part because of its relevance to the XSD Schema specification.

XPath Data Model

An XML document, when it exists as a text file, is simply a sequence of characters. An XML parser processes the stream of characters that it encounters and, if the document

is well-formed XML, is able to extract from the sequence of characters a number of logical tokens that represent elements and other parts of the source XML document. Just as a human being who understands XML can read an XML document and extract the logical structure that is present in the document, so an XML parser can determine a logical structure and model that structure as XPath nodes.

XPath models an XML document as a hierarchy of nodes. Frequently the hierarchy of nodes is referred to as a tree but, technically speaking, an XSLT or other XPath processor need not implement the document as a tree.

An XPath processor can compute a string-value for any node. Some types of nodes (element nodes, for example) also have names.

The XPath data model also fully supports the use of XML namespaces. Thus the name of an XPath node is modeled (for those nodes that have a name) as a pair consisting of a local part and a namespace URI. In practice, the name is typically expressed in a source document as a QName, with a namespace prefix (declared in a namespace declaration) being used as an indicator of the namespace URI with which it is associated.

Thus if we had an element like the following, the namespace prefix is "xmml", the local part is "services", and the namespace URI is "http://www.xmml.com/Schemas/".

```
<xmml:services xmlns:xmml="http://www.xmml.com/Schemas/">
  <!-- Content goes here. -->
</xmml:services>
```

Remember that although humans find it easier to use the namespace prefix, the XPath processor makes use of the namespace URI.

An XPath processor examines the hierarchy of nodes to determine if a location path matches one or more of the nodes. The resulting node set may contain many nodes, a single node, or may be an empty set.

The In-Memory Tree

When an XML document is being processed, there are two in-memory trees: the source tree and the result tree. This was shown schematically in Figure 2.2. The serialized XML source document is converted to a source tree; then that source tree is used to create a result tree when using XSLT. Finally the result tree is converted to the output document.

We will first look at the source tree.

The Source Tree

Let's start with a very simple XML source document and look at how, schematically, it will be represented in memory. It is this tree-like representation of the XML source document that is processed by an XSLT/XPath or XPointer processor. An XPath processor cannot directly act on the character sequences which exist in a well-formed or valid XML document, but depends on its being made available in memory as a tree structure, consisting of nodes.

Listing 3.1 is a simple source XML document.

```
<?xml version='1.0'?>
<?xml-stylesheet href="SomeXSLT.xsl" type="text/xsl"?>
<!-- This is a comment. It will have a node. -->
<Book Title="I did it my way!"/>
```

Listing 3.1 A Simple XML Document with Attributes and a Comment (Book.xml).

The schematic of how that document is represented in memory is provided in Figure 3.1.

The root node must be present in the XPath in-memory hierarchy. All nodes in the representation of an XML document are descendants of the root node.

It may surprise you that there is only one processing instruction node in the diagram. Why isn't there a processing instruction node for the XML declaration? If it did surprise you, then you need to remember that, technically speaking, the XML declaration is not a processing instruction, despite the fact that it uses the same initial "<?" characters and closing "?>" characters as are used in a processing instruction. In fact, an XML declaration is not represented in any way in an XPath in-memory tree.

In Figure 3.1 the root node is the parent of three nodes. Each of those three nodes is in the child axis. The first child node is the processing instruction node, which represents the processing instruction.

```
<?xml-stylesheet href="SomeXSLT.xsl" type="text/xsl"?>
```

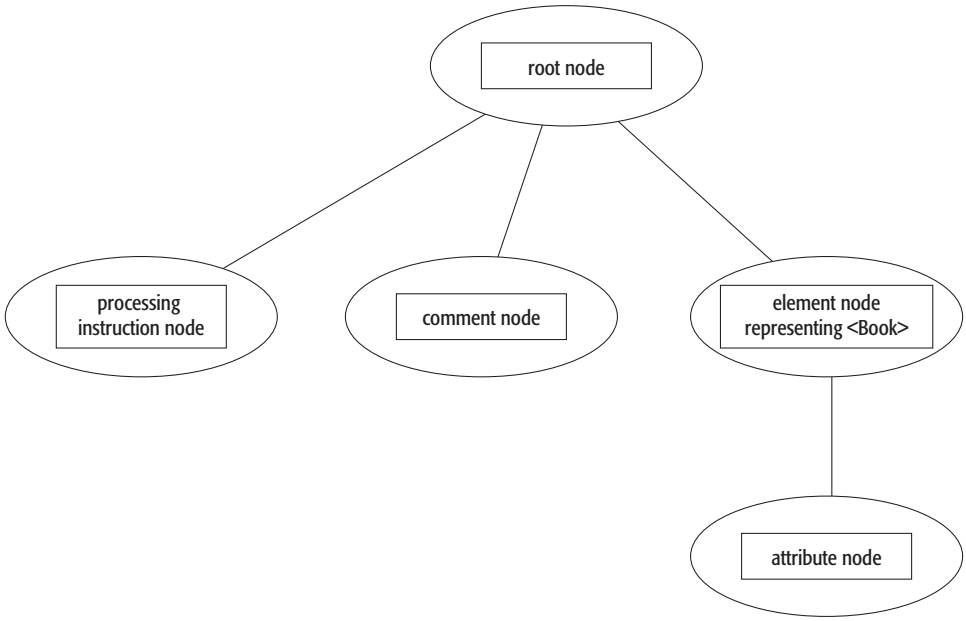


Figure 3.1 Schematic Representation of the Source Tree.

The second child node is a comment node, which represents the comment in the brief source document. The third child node of the root node is the element node representing the <Book> element in the source document.

The processing instruction node, the comment node, and the element node would be accessed using the XPath child axis.

The <Book> element in the source document possesses a Title attribute. The Title attribute is represented in the in-memory tree by an attribute node. If the element node representing the <Book> element is the context node, then the location path to describe how to get to the attribute node is expressed as follows in the unabbreviated XPath syntax.

```
attribute::Title
```

In other words, to get to the attribute node from the element node, the XPath processor must traverse the attribute axis. The Title attribute node is *not*, in XPath terminology, a child node of the element node that represents the <Book> element.

So, that's how the in-memory tree seems to look. But if we apply the stylesheet shown Listing 3.2 to the source document, we see a slightly different story.

The output from applying the stylesheet is shown in Figure 3.2.

Thus we can see that there is an invisible namespace node on the <Book> element. All namespace-aware XML documents are associated implicitly with the namespace http://www.w3.org/XML/1998/namespace.

TEAMFLY

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>namespace nodes</title>
</head>
<body>
<xsl:apply-templates select="Book"/>
</body>
</html>
</xsl:template>

<xsl:template match="Book">
<xsl:for-each select="namespace::*">
<p>For node named <xsl:value-of select="name()"/> the namespace node is
<xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Listing 3.2 An XSLT Stylesheet to Transform the Source XML Document (SomeXSLT.xsl).

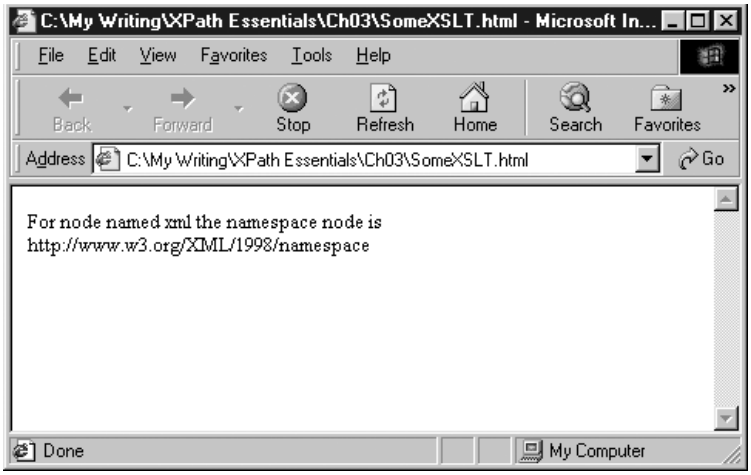

C:\My Writing\XPath Essentials\Ch03\SomeXSLT.html - Microsoft In...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch03\SomeXSLT.html
Go
For node named xml the namespace node is
http://www.w3.org/XML/1998/namespace
Done
My Computer

Figure 3.2 Demonstrating the Presence of an XML Namespace Node.

Figure 3.3 more completely represents the in-memory tree for our simple source document.

If we create a slightly more complex source document (see Listing 3.3) with a visible namespace prefix, we can examine the issue of namespace nodes a little more closely.

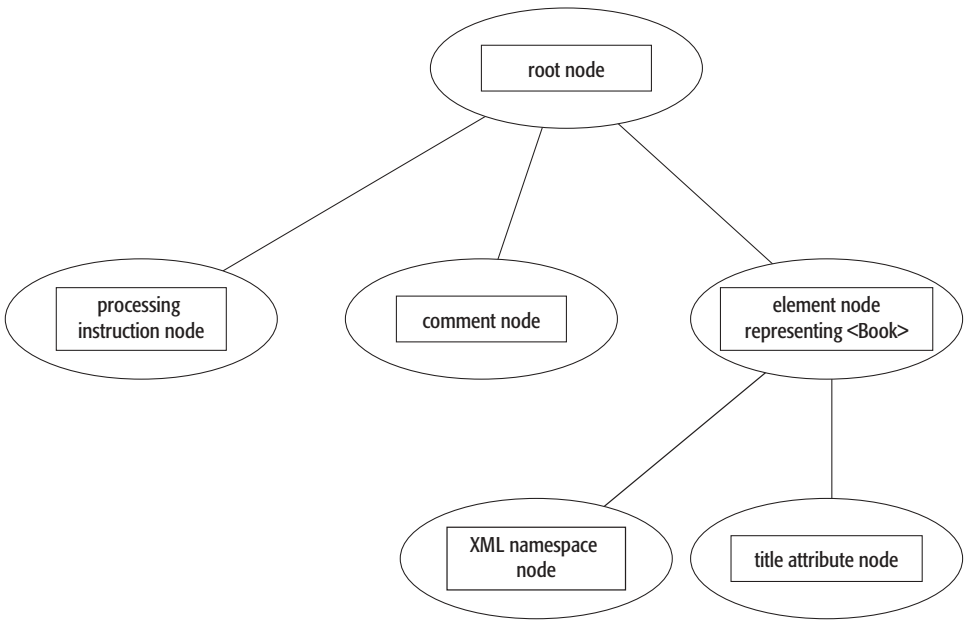


Figure 3.3 In-memory Tree with XML Namespace Node.

```
<?xml version='1.0'?>
<XMML:Publications xmlns:XMML="http://www.xmml.com/Publications/">
<XMML:Book XMML:Title="I did it the XMML.com way!"/>
</XMML:Publications>
```

Listing 3.3 An XML Source Document Which Uses Namespaces (XMMLPublications.xml).

The stylesheet in Listing 3.4 can be used to display the namespace nodes for the <XMML:Publications> element.

As you can see in Figure 3.4, which shows the HTML output from the transformation, there are now two namespace nodes on the element root, which in this case is the <XMML:Publications> element. One of the namespace nodes is the implicit XML namespace of http://www.w3.org/XML/1998/namespace and the other is the namespace URI explicitly declared in the source document and in the stylesheet.

Thus, if we represent the source document visually, we can see, as in Figure 3.5, all the namespace nodes that are related to the source document.

Having looked at the source tree, let's move on to look at the result tree.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
       xmlns:XMML="http://www.xmml.com/Publications/"
       >

<xsl:template match="/">
<html>
<head>
<title>Showing the nodes in a document which includes namespaces</title>
</head>
<body>
<xsl:apply-templates select="XMML:Publications"/>
</body>
</html>
</xsl:template>

<xsl:template match="XMML:Publications">
<p>The element node representing &lt;<xsl:value-of select="name()"/>&gt;
  has the following namespace nodes -</p>
<xsl:for-each select="namespace::*">
<p>For node named <xsl:value-of select="name()"/> the namespace node is
<xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 3.4 A Stylesheet to Transform the Source Document Which Contains Namespaces (XMMLPublications.xsl).

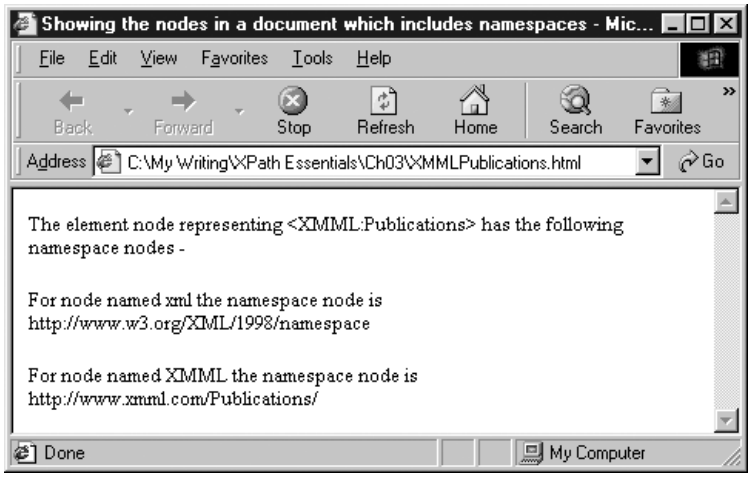

Showing the nodes in a document which includes namespaces - Mic...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch03\XMMLPublications.html
Go
The element node representing <XMML:Publications> has the following namespace nodes -
For node named xml the namespace node is http://www.w3.org/XML/1998/namespace
For node named XMML the namespace node is http://www.xmml.com/Publications/
Done
My Computer

Figure 3.4 Namespace Nodes on the XMML:Publication Element.

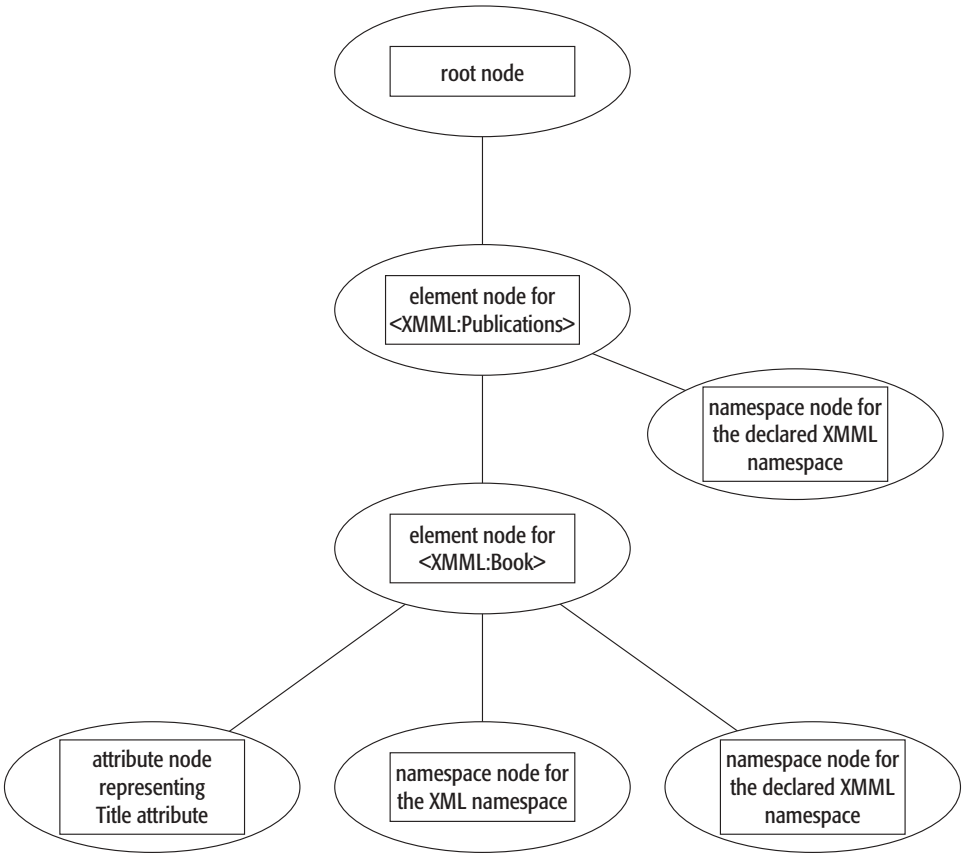

root node
element node for <XMML:Publications>
namespace node for the declared XMML namespace
element node for <XMML:Book>
attribute node representing Title attribute
namespace node for the XML namespace
namespace node for the declared XMML namespace

Figure 3.5 Schematic of the In-Memory Tree of the Source Document with Namespaces.

The Result Tree

An XSLT transformation, typically using XPath location paths as patterns, creates a result tree, not (primarily) a result document.

The result tree, just as the source tree, consists of a hierarchy of nodes. The result tree also has a root node at the head of the hierarchy, and all other nodes in the result tree are descendants of that root node.

NOTE The result tree is an intermediate step in the creation of a result document.

The result tree is, in effect, a representation of another XML document since the XPath tree only applies to the representation of such XML documents. The conversion of the result tree to a serialized form follows the creation of the result tree.

Thus, if we have a simple XML source document like that in Listing 3.5 and we wish to convert its structure so that the content of the <Title> element becomes the value of a <Title> attribute, we can carry out such a transformation using the stylesheet in Listing 3.6.

```
<?xml version='1.0'?>
<MyBook>
<Title>XPath Essentials</Title>
</MyBook>
```

Listing 3.5 A Simple Description of a Book in XML. (MyBook.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<xsl:apply-templates select="MyBook"/>
</xsl:template>

<xsl:template match="MyBook">
<xsl:copy>
<xsl:attribute name="Title">
XPath Essentials
</xsl:attribute>
</xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Listing 3.6 A Stylesheet to Modify the Structure of the Source Document (MyBook.xsl).

```
<?xml version="1.0" encoding="utf-8"?>
<MyBook Title="XPath Essentials"/>
```

Listing 3.7 The Output from the Stylesheet Shown in Listing 3.6 (MyBook2.xml).

The appearance of the output document, MyBook2.xml, shown in Listing 3.7 (formed at a later stage by serializing the result tree), is shown in Figure 3.6.

The result tree is created step by step as the XSLT transformation processes the source document.

As the main template is instantiated, a root node is created as the head of the hierarchy for the result tree.

The <xsl:apply-templates> element passes control to the <xsl:template> element that matches an element node that represents a <MyBook> element. That same element node becomes the context node.

The <xsl:copy> element causes an element node to be added to the result tree. Since the only part of the result tree at that time is the root node, the element node representing the <MyBook> element is added as a child of the root node.

The <xsl:attribute> element causes an attribute node to be added to the result tree. There is a suitable element node available for it to be associated with (the element node representing the <MyBook> element); therefore, it is added to the result tree with the element node representing the <MyBook> element as its parent node.

Thus, the result tree from this very simple transformation is represented in Figure 3.7.

If you compare the source tree in Figure 3.8 with the result tree in Figure 3.7, you can see that an element node (and its corresponding namespace node) have been removed from the source tree and an attribute node has been created to replace the removed element node.

Documents and Trees

The source and result trees are important parts of a transformation but are not the whole story. We also need to think about the source document and result document.

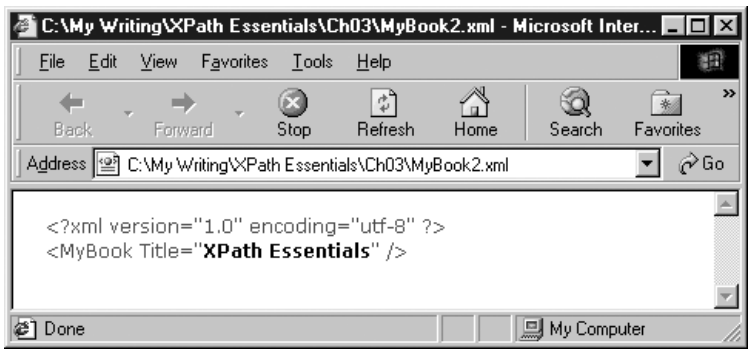


Figure 3.6 The Result of the XSLT Transformation.

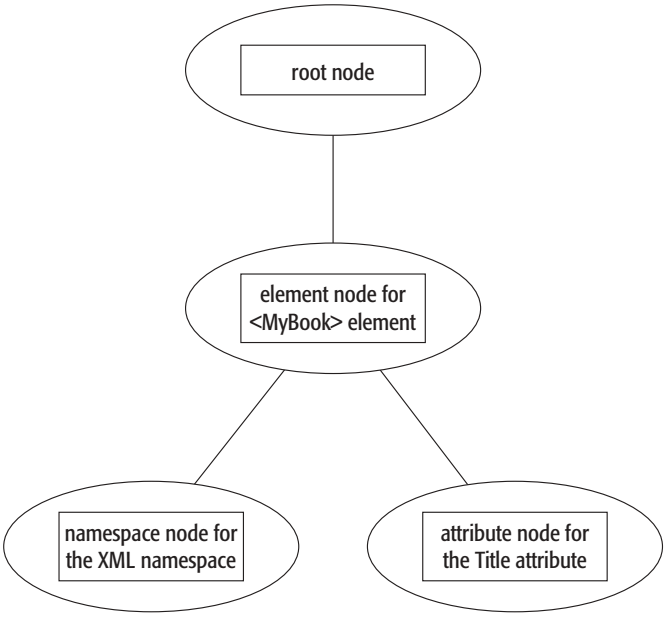


Figure 3.7 Schematic of the Result Tree from an XSLT Transformation.

In Figure 3.9 the source document and result document are represented as rectangles and the source tree and result tree are represented as a hierarchy of nodes.

Within the process of an XSLT transformation, step 1 is to parse the source XML document from a sequence of characters (which happen to conform to the syntax requirements of XML 1.0) to a hierarchy in memory, which is termed the source tree.

Step 2 is the XSLT transformation where the source tree is transformed, as defined by an applied XSLT stylesheet, to the result tree.

Step 3, in this schema, is the conversion of the result tree into a result document (also known as an output document), which may be an XML or HTML document, or may be some form of text document.

Of the three steps, an XSLT processor is obliged by the specification to carry out only Step 2. However, a typical XSLT processor will also make use of an XML parser to carry out Step 1 and will be capable of outputting the result tree in an appropriate serialized format.

Node Types

XPath has seven types of nodes, which correspond to elements, attributes, etc., within an XML document.

The node types available in XPath 1.0 are

- Root node
- Element nodes

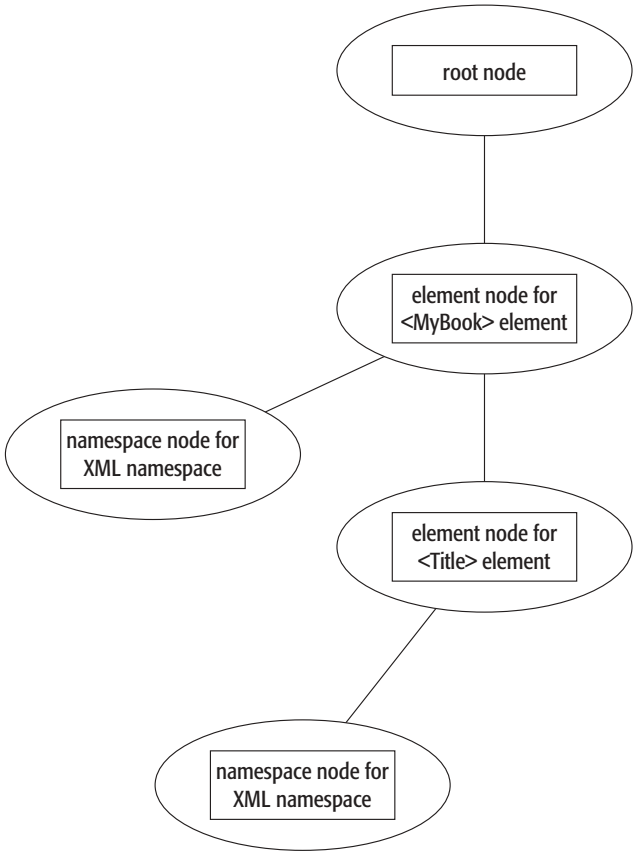


Figure 3.8 Schematic of the Source Tree.

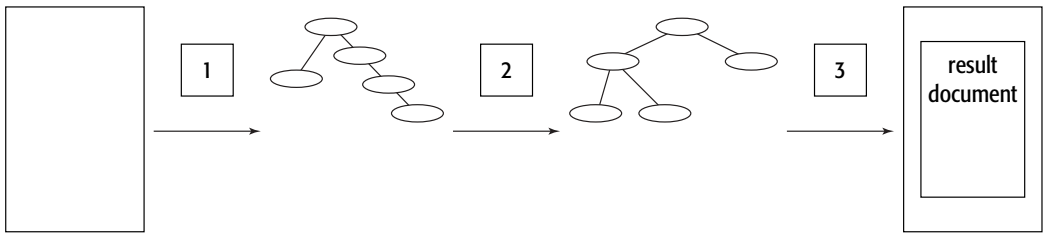


Figure 3.9 Schematic of an XSLT Transformation.

- Attribute nodes
- Comment nodes
- Namespace nodes
- Processing instruction nodes
- Text nodes

In the following sections we will take a closer look at each of the node types.

Root Node

Each XML document on which XPath operates has one and only one root node. The root node occurs at the "root" of the tree, in other words, at the top spot in the treelike hierarchy in which the XML document is represented in memory.

The *element root* (also known as the *document element*) of an XML document is represented by an element node which is a child node of the root node. A root node may also have comment nodes and processing instruction nodes as children. These nodes represent any comments and processing instructions that occur either in the prolog of an XML document (that is, before the document element in document order) or after the closing tag of the document element (a position sometimes informally referred to as the *epilog*). Note that the root node does not have a child that represents the XML declaration, if present, nor the Document Type Declaration, if present. In fact XPath provides no representation of either the XML declaration or the Document Type Declaration in the XPath tree.

A root node has a string value that equals the concatenation of the string-value of all text node descendants of the root node.

The root node does not have an expanded name.

Element Nodes

In the XPath tree there is an element node for each element in the source document.

An element node may have children that may be any or all of four node types: *element nodes*, *comment nodes*, *processing instruction nodes*, and *text nodes*. Attribute nodes and namespace nodes are not, in XPath terminology, children of the element node with which they are associated.

Each element node has an expanded name. The value of the expanded name is calculated by expanding the QName of the element type name that is contained in the start or end tag according to the rules set out in the Namespaces in XML Recommendation (located at www.w3.org/TR/REC-xml-names). An expanded name thus has two parts: the *namespace URI* (also called the *namespace name*) and the *local part*.

Thus if we had an element with the QName

```
<MyPrefix:MyElement xmlns=" http://www.AWonderfullyLongAndAwkwardName
.com/Schemas/">
```

then the expanded-name replaces the namespace prefix with the namespace URI, "http://www.AWonderfullyLongAndAwkwardName.com/Schemas/". The local part remains unaltered.

NOTE **Although the structure to express a QName within an XML element is made clear, the precise structure of an expanded-name is more ambiguous. It is clear that it has two parts but exactly how they would be written is not stated.**

The namespace URI of the element's expanded-name will be null if the QName has no prefix and there is no applicable default namespace.

The string-value of an element node is arrived at by concatenating all the text node descendants of the element node, in document order. Thus, for example, the string-value of the node representing the <Chapter> element would be the natural English sentence because of the document order—the order in which the <SomeText> elements occur in the XML document.

```
<Chapter>
<SomeText>The string-value is arrived at by concatenating<SomeText>
<SomeText>the string-value of text nodes which are descendants of the
element.</SomeText>
</Chapter>
```

An element node may have a unique identifier, an ID, which is an attribute associated with the element that has been declared in an accompanying DTD as being of type ID. It is not permitted that two elements in the same document have the same ID. If that occurs, then the first element in document order is treated as having the said unique ID and the second element, which *appears* to have the same unique ID, is treated as if it lacked a unique ID. Of course, an ID attribute may only occur where an attribute has been declared to be of type ID in an accompanying DTD. In XPath 1.0 the arrival of the XML (XSD) Schema language is not acknowledged since the XPath Recommendation was finalized about 18 months before XML (XSD) Schema became a Recommendation in May 2001.

Attribute Nodes

Each element node that represents an element in the source XML document has a set of attribute nodes, which will be an empty set if the element has no attributes. The element node is the parent of each attribute node. However, seemingly perversely, the attribute node is not a child of the element node. This makes sense when one realizes that the child axis has a default node type of element and that the child axis does not contain attribute nodes. There is a separate attribute axis to associate attribute nodes with their parent element nodes.

NOTE **In this respect XPath differs from the Document Object Model, which does not consider that an element node is the parent of an attribute node that is associated with it.**

An attribute node only has one parent node. Element nodes never share attribute nodes. Thus in the tree which represents the following XML fragment there would be two element nodes, one for each <Chapter> element, and there would be two separate

attribute nodes representing the two distinct attributes, despite the attributes sharing the same name.

```
<Chapter number="1">Some chapter title</Chapter>
<Chapter number="2">Some other chapter title</Chapter>
```

Attribute nodes may be present in the tree hierarchy despite not having been explicitly expressed in the XML document, if the attribute is declared in an accompanying DTD. Presumably at some future date, XPath processors will, in due time, recognize XPath 2.0 and will similarly recognize attributes defined in an XSD Schema document.

The xml:lang and xml:space attributes have a scope that may extend over many elements in a source document. However, in the tree representation of that document an attribute node is present only for the element node on which the attribute in question was present in the respective start tag.

An attribute has an expanded name. In the following example, the MyPrefix:MyAttribute is the QName. That QName is expanded to include the local part, MyAttribute, and the namespace URI, http://www.AWonderfullyLongAndAwkwardName .com/Schemas/".

```
<MyPrefix:MyElement xmlns=" http://www.AWonderfullyLongAndAwkwardName
  .com/Schemas/" MyPrefix:MyAttribute="Some value">
```

Each attribute node has a string-value. The string-value is the normalized value of the attribute as specified in the rules for normalization in the XML 1.0 Recommendation (see Chapter 3.3.3 of www.w3.org/TR/2000/REC-xml-20001006).

No attribute node exists for those attributes in an XML document that are namespace declarations.

TEAMFLY

Comment Nodes

There is a comment node in the XPath hierarchical representation for each comment present in the XML source document, with the exception of any comments that may be contained in the Document Type Declaration. The absence of any comment node for comments present in the Document Type Declaration seems to arise from the failure of XPath to represent the Document Type Declaration at all.

The string-value of a comment node is the string that occurs between the "<!—" and "—>", with those delimiters being excluded from the string-value.

A comment node does not have an expanded name.

Namespace Nodes

Each element in an XML document has an associated set of namespace nodes. A namespace node exists for each namespace prefix that is in scope for the element, which includes the "xml" prefix (which is implicitly declared in documents that comply with the Namespaces in XML Recommendation) and the default namespace if one is in scope for the element.

The element node is the parent of each of these namespace nodes. However, a namespace node is not a child of its parent node. There is a namespace axis (which has

principal node type of namespace node) in XPath that is distinct from the child axis, whose principal node type is the element node. Thus in a document that uses multiple namespaces, the number of namespace nodes associated with individual element nodes can be considerable.

Strangely, although an attribute node may have a namespace prefix and be associated with a namespace URI, it does not possess a separate namespace node.

A namespace node has an expanded name, but it has some unusual features. The local part is the namespace prefix. Where the namespace node is for the default namespace the local part is empty. The namespace URI is always null.

The string-value of a namespace node is the namespace URI that is bound to the namespace prefix.

Processing Instruction Nodes

There is a processing instruction node in the XPath hierarchical representation for each processing instruction in the source XML document, except that any processing instructions that may occur within the Document Type Declaration have no corresponding node in the XPath hierarchy. This, in part, arises from the absence of recognition of the Document Type Declaration throughout XPath 1.0.

A processing instruction node has an expanded name. The local part of the expanded name is the target for the processing instruction. The namespace URI is always null.

The string-value of a processing instruction node is the part of the processing instruction following the target, including any whitespace, but excluding the final "?>".

NOTE **Remember that, technically speaking, the XML declaration is not a processing instruction despite the superficial similarity of an opening "<?" and closing "?>". An XML declaration, unlike a processing instruction, has no target. Nor is an XML declaration represented in the XPath tree hierarchy.**

Text Nodes

Character data in the XPath representation of an XML document is grouped into text nodes. Text nodes are created in such a way that as much text data is placed on each text node as possible with the result that the number of text nodes is minimized; in particular, a text node may not have a sibling text node.

The issues relating to CDATA sections merit mention. If a CDATA section exists in a source document then a text node is created in the hierarchy that contains the character data between the "<![CDATA[" and the final "]]>", but not including the delimiters themselves. However, there is nothing that indicates that the text node was created from a CDATA section and therefore no guarantee that a CDATA section in a source document will also automatically be serialized as a CDATA section in a result document.

The string-value of a text node is the character data that it contains. A text node must have at least one character of data, and therefore the string-value of a text node cannot be the zero length string.

Table 3.1 Node Types

NODE TYPE	NAMED?
Root node	No
Element node	Yes
Attribute node	Yes
Comment node	No
Namespace node	Yes
Processing instruction node	Yes
Text node	No

Node Names

Some types of node in XPath have names, but not all do. Table 3.1 summarizes the situation.

Having taken a look at the XPath model of an XML document, let's take a brief look at the Document Object Model.

Document Object Model

The Document Object Model (DOM) Level 1 was designed to provide an object model for both HTML and XML documents. DOM was designed in order to allow programming and scripting languages to dynamically access and update the content and structure of an HTML or XML document.

NOTE For further general information about the Document Object Model and its ongoing development, consult the DOM page on the W3C Web site at www.w3.org/DOM/.

Thus, DOM is explicitly intended to provide an application programming interface to parts of an XML (or HTML) document. By contrast XPath has no API. DOM was designed to allow the API to be manipulated by any programming language, and the DOM interfaces were specified in Interface Definition Language. The specification also defined language bindings for Java and for ECMAScript (the standards-based development of JavaScript).

DOM Level 1 Recommendation was issued by W3C in October 1998. The Recommendation is located at www.w3.org/TR/1998/REC-DOM-Level-1-19981001. Parts of that Recommendation were HTML-specific. The DOM Core and extended interfaces applied to XML. DOM Level 1 provided interfaces only for accessing and manipulating content and structure. Level 1 consciously did not support the following:

- Structure model for the internal subset and the external subset
- Validation against a schema
- Control for rendering documents via stylesheets
- Access control
- Thread safety
- Events

The DOM Level 2 Recommendation was issued as a series of modules. The specifications of the individual modules are accessible from the W3C Web site.

NOTE For XPath root nodes and element nodes, the string-value of the node is not the same thing as the string returned by the DOM nodeValue method.

DOM Level 1 and Level 2 seemed to be on a separate path from XPath, but with the advent of DOM Level 3, there is an emerging indication of improving interoperability between the specifications.

DOM Level 3 XPath

The W3C DOM Working Group is developing a specification that defines how to access a DOM tree using XPath syntax. DOM Level 3 XPath is one of several DOM Level 3 modules. The DOM Level 3 XPath module builds on the DOM Level 3 Core Module. If you wish to make use of DOM Level 3 XPath, you will also need to be familiar with the Core Module, which is located at www.w3.org/TR/DOM-Level-3-Core.

The objective of the DOM Level 3 XPath proposal is to overcome the mismatches between the object models of XPath 1.0 and DOM levels 1 and 2. Let's first take a closer look at those.

NOTE A DOM Level 3 Working Draft has been issued by W3C, which addresses issues relating to the interoperability of DOM and XPath. The first Working Draft was issued on June 18, 2001, and is located at www.w3.org/TR/2001/WD-DOM-Level-3-XPath-20010618. Any updates to that specification will be located at www.w3.org/TR/DOM-Level-3-XPath.

DOM and XPath Object Models

There are several differences in approach, as well as in terminology, which potentially cause difficulty translating between a DOM and XPath model.

Text Nodes

One significant difference between XPath and DOM is how they handle text nodes.

Take the following simple rhyme in Listing 3.8 as an example. XPath would handle this so that the <Mary> element had a single text node child. DOM would be expected

```
<?xml version='1.0'?>
<Mary>
Mary had a little lamb
Its fleece was white as snow
<![CDATA[And everywhere that Mary went]]>
The lamb was sure to go
</Mary>
```

Listing 3.8 A simple nursery rhyme in XML (Mary.xml).

to split the text to correspond to the presence of the CDATA section, since DOM has both Text and CDATASection interfaces.

DOM also contains a splitText method associated with the Text interface, which, if a node originally corresponded to an XPath single text node, would result in an anomalous position of having two sibling nodes, each containing text in the DOM model.

The proposals for DOM Level 3 Core add a wholeText attribute to the Text interface, allowing the whole text for logically adjacent text nodes to be retrieved. This presents a DOM approximation of how XPath handles a single text node.

Namespace Nodes

As I indicated earlier in the chapter, XPath duplicates a namespace declaration across each element to which it applies, by adding a namespace node to each corresponding element node. In contrast, DOM does not duplicate namespace nodes in that way, but maintains the declaration of namespaces without duplicating them on each element node. The DOM `OwnerElement` is different from the individual element nodes in XPath, which are the parents of namespace nodes.

The precise means to solve this disparity in approach is not yet clear, but the fact that the DOM Working Group has issued a Working Draft focused solely on the issue of XPath interoperability bodes well for the future resolution of these discrepancies.

DOM has no declaration node corresponding to the always-present XML namespace (www.w3.org/XML/1998/namespace). To achieve compatibility with XPath, some representation of that namespace node (or separate ones for each relevant element node) needs to be created.

Another issue is that the uniqueness and document order of namespace declaration nodes is different from in-scope namespace nodes, producing a different order and number of nodes in the result set where the namespaces nodes of multiple elements are requested.

Among the other issues yet to be resolved are how and to what degree the DOM Level 3 XPath should attempt a complete solution of compatibility with XPath 1.0 (the subject of this book) or how much they should skip ahead and aim to have a solution in place for the arrival of XPath 2.0 (and XQuery 1.0) which, as final specifications, are likely to lie some way off.

NOTE **The DOM Level 3 Core and DOM Level 3 XPath specifications are both at Working Draft status; therefore, it is very possible that significant changes may take place in relation to the proposed solutions to the disparities between XPath and DOM data models. If you require to apply these proposed solutions, then be sure to check the latest versions at the URLs given earlier.**

Document Type Declaration

The DOM specification represents the Document Type Declaration via a `DocumentType` interface. XPath lacks any means to express the existence or content of the Document Type Declaration.

Conclusion

Let's move back to the general solution of the DOM/XPath compatibility issues.

DOM Level 3 XPath proposes an XPathEvaluator interface. Currently it is proposed that the XPath evaluator interface should have four methods: evaluateAsBoolean, evaluateAsNodeSet, evaluateAsString, and evaluateAsNumber, corresponding to the four data types in XPath 1.0.

This whole issue of resolving how to promote interoperability between DOM and XPath is a particularly fluid one. It is clearly not straightforward, given the current characteristics of XPath 1.0 and the evolving discussion of the interaction between the requirements documents for XPath 2.0, XSLT 2.0, and XQuery 1.0 (discussed in Chapter 14, "XPath and XQuery"). Final specifications for these are some considerable way off at the time of writing, but I will discuss the future of XPath and its likely relationship to XQuery 1.0 in Chapter 14.

Having looked at the W3C's 1998 effort at representing an XML document (the DOM), let's take a close look at the model that is likely to strongly influence, and possibly dominate, the future of the modeling of XML documents, including the future for XPath.

XML Information Set

The XML Information Set provides a data model for much, but not necessarily all, of an XML document. The W3C uses the term *abstract data set* to describe the XML Information Set. The XML Information Set is expressed in terms of *information items*. An explicit aim of the XML Information Set is to provide a consistent set of definitions—contrasting the disparities between XPath and DOM—for the information contained in a well-formed XML document.

NOTE **The XML Information Set specification is, at the time of writing, at Candidate Recommendation status, and the specification is located at www.w3.org/TR/2001/CR-xml-infoset-20010514. Any updates since that version will be located at www.w3.org/TR/xml-infoset. If the May 2001 Candidate Recommendation is displayed at the latter URL, that indicates that no further versions of the specification have been issued in the interim.**

One word of caution about the XML Information Set: It does not attempt to be exhaustive, but simply to address issues that are most likely to be useful. In a sense, the XML Information Set makes explicit what XPath did without actually stating it. XPath, for example, omitted any representation of the XML declaration and the Document Type Declaration, presumably because a representation of those aspects of the source XML document was perceived to be of little value in the context of the then-perceived uses of XPath.

An XML document has an information set if it is well-formed and if it satisfies the namespace constraints outlined in the XML Information Set specification. Specifically the namespace constraints are that a colon character may only be used in the source document in the manner permitted by the Namespace in XML Recommendation (see www.w3.org/TR/1999/REC-xml-names-19990114) and that any URI reference must be an absolute URI. If a relative URI is used in a namespace declaration, then the document does not have a defined XML Information Set!.

NOTE **An external entity which is not, in itself, well-formed does not have an information set, whether or not the resulting document would be well-formed when the entity had been expanded in its source document. If it is impossible to access an external entity, an "unexpanded entity reference" information item is included within the information set of the XML document with which the entity is associated.**

The XML Information Set specification only describes the creation of information sets by the parsing of source XML documents. It does not define the information set created by, for example, creating an information set using the DOM or as a result of an XSLT transformation.

The XML information set consists of a set of "information items," which we will now discuss.

Information Items

The XML information set for a well-formed XML document consists of a set of information items, which represent parts of the document. In practice each information set will consist of a document information item and several others.

Each information item has an associated set of named properties. In XML Information Set notation the properties are contained in [square brackets], like so. The use of square brackets is identical to those used in XPath predicates. When XPath and the infoset are used together more often, it seems that a potential source of confusion will exist.

Several information items have a [baseURI] or a [declarationbaseURI] property. These properties are computed according to the XML Base Recommendation (located at www.w3.org/TR/2001/REC-xmlbase-20010627/).

Some XML Information Set information item properties may take the value of "unknown" or "no value." The "unknown" and "no value" values are not the same, nor are they the same as the empty string, the empty list, or an empty node set. The XML Information Set specification does not use the term "null."

The XML Information Set does not specify or require a particular manner of implementation or a particular set of interfaces. In that respect, the infoset resembles XPath in lacking an API. The DOM, on the other hand, includes the presentation of an API as a foundational aspect.

The specification makes it clear that the term "information set" is similar in meaning to a "tree" and that the term "information item" is similar in meaning to a "node." The specification goes on to make it clear that there is no straightforward universal one-to-one mapping from an XML infoset information item to a DOM node or an XPath node.

NOTE The relationship between XPath 1.0 and an early (1999) draft of the XML Information Set is described in Appendix B of the XPath Recommendation at www.w3.org/TR/xpath#infoset.

In the following sections I will briefly list the information items in the XML Information Set and briefly describe their properties. As you read these sections, you will see that the model in the XML Information Set is substantially different from the XPath model.

Document Information Item

In each information set there is one (and only one) document information item. Each of the other information items contained in the representation of a document are accessible from the properties of the document information item and from the properties of information items that are accessible from the properties of the document information item.

Thus, in a sense, the properties of the document information item have some similarities to the axes in the XPath data model.

NOTE The XML Information Set makes use of parts of the XML 1.0 Recommendation that are not prominent or present in XPath. If you need to refresh your memory, consult either Chapter 1 of this book or the XML 1.0 Recommendation (located at www.w3.org/TR/2000/REC-xml-20001006).

The document information item has nine properties:

- [children]
  An ordered list of child information items in document order. The list of children contains exactly one element information item. In addition there is a processing instruction information item and a comment information item for each processing instruction or comment that exists outside the document element. If there is a document type declaration in the document, then there is a corresponding document type declaration information item in the list in this property. Note the dissimilarity from XPath, which does not represent a document type declaration.

- [document element]
  The element information item that corresponds to the document element.

- [notations]
  An unordered set of information items, one for each notation declared in a DTD, if one exists. Note that no mention is made of XSD Schema in this context.
- [unparsed entities]
  An unordered set of unparsed entity information items, one for each unparsed entity declared in a DTD, where one exists.
- [base URI]
  The base URI of the document entity.
- [character encoding scheme]
  The character encoding scheme in which the document entity is expressed.
- [standalone]
  An indication of the standalone status of the document derived from the optional XML declaration.
- [version]
  Derived from the XML declaration.
- [all declarations processed]
  Strictly speaking, not part of the information set. It indicates whether or not all of the DTD has been processed.

As you can see, the model of the XML Information Set is distinctly different from that of XPath 1.0.

Element Information Item

There is an element information item for each element in the source document, much as there is an element node in XPath. However, the properties of the element information item differ significantly from the characteristics of an XPath element node.

An element information item has nine properties:

- [namespace name]
  The namespace name (namespace URI) of the element information item, if it belongs to a namespace.
- [local name]
  The local part of the element type name.
- [prefix]
  The namespace prefix.
- [children]
  An ordered list of child information items in document order. The information items that may be present are the element information item, the processing instruction information item, unexpanded entity reference information item, the character information item, and the comment information item, corresponding to similarly named constructs in the source document.

- [attributes]
  An unordered set of attribute information items. Namespace declarations are not included in this set.

- [namespace attributes]
  An unordered set of attribute information items corresponding to each namespace declaration of the element information item. Together the [attributes] and [namespace attributes] properties list all attributes of the element information item.

- [in-scope namespaces]
  An unordered set of namespace information items, one for each namespace in scope for the element.

- [base URI]
  The base URI of the element.

- [parent]
  The document or element information item that contains this element information item in its [children] property.

Note the major difference between XPath with its homologated text node and the XML Information Set where each character has a corresponding information item. Note also the different approach from XPath in the handling of namespace-related information.

Attribute Information Item

There is an attribute information item for each attribute on each element in the document, whether the attribute is explicitly expressed in the XML source document or is declared in an accompanying Document Type Definition.

An attribute information item has eight properties:

- [namespace name]
  The namespace name (namespace URI), if any, of the attribute

- [local name]
  The local part of the attribute name

- [prefix]
  The namespace prefix

- [normalized value]
  The attribute value normalized according to the rules in Chapter 3.3.3 of the XML 1.0 Recommendation

- [specified]
  A flag indicating whether the attribute was defined in a DTD or was present in the start tag of the owner element

- [attribute type]
  The type declared for the attribute in an accompanying DTD, if any. The permitted values are ID, IDREF, IDREFS, ENTITY, NMTOKEN, NMTOKENS, NOTATION, CDATA, and ENUMERATION

- [references]
  If the attribute type is IDREF, IDREFS, ENTITY, ENTITIES, or NOTATION the value of the references property is an ordered list of the element information items, unparsed entity information items, and notation information items referred to in the attribute value, in the order that they occur

- [owner element]
  The element information item that contains this attribute information item in its [attributes] property.

Processing Instruction Information Item

There is a processing instruction information item in the infoset for each processing instruction in the source XML document. The XML declaration in the source document and the text declaration in any external parsed entities are not considered processing instructions.

A processing instruction information item has five properties:

- [target]
  A string (an XML name) representing the target part of the processing instruction

- [content]
  A string representing the content of the processing instruction (less the target and any whitespace)

- [base URI]
  The base URI of the processing instruction

- [notation]
  The notation information item named by the target, if there is one

- [parent]
  The document information, element information item, or document type declaration information item in whose [children] property the processing instruction appears

Unexpanded Entity Reference Information Item

An unexpanded entity reference information item exists for any external entity that cannot be or has not been expanded.

An unexpanded entity reference information item has five properties:

- [name]
  The name of the entity that was referenced, but not expanded

TEAMFLY

- [system identifier]
  The system identifier of the entity
- [public identifier]
  The public identifier of the entity, normalized as described in Chapter 4.2.2 of the XML 1.0 Recommendation.
- [declaration base URI]
  The base URI relative to which the system identifier should be resolved
- [parent]
  The element information item that contains this unexpanded entity reference information item within its [children] property

XPath has no equivalent of this information item.

Character Information Item

There is a character information item for each character that appears in the document, whether literally as a character reference or in a CDATA section. From the viewpoint of the XML Information Set, each character in the source document is a logically separate information item, but the XML Information Set specification places no restrictions on how an XML application may find it appropriate to aggregate such character information items.

A character information item has three properties:

- [character code]
  The ISO 10646 code for the character
- [element content whitespace]
  A Boolean value indicating whether or not the character is whitespace
- [parent]
  The element information item in whose [children] property this character information item exists

Comment Information Item

There is a comment information item in the information set for each comment in the original document.

A comment information item has two properties:

- [content]
  The string between the opening "<!—" and closing "—>" delimiters
- [parent]
  The document information item or element information item within whose [children] property the comment information item exists

Document Type Declaration Information Item

An information set may contain zero or one document type declaration information items, depending on whether or not the original document contains a document type declaration. Note that entities and notations are specified in the properties of the document information item, not in the properties of the document type declaration information item.

A document type declaration information item has four properties:

- [system identifier]
  The system identifier of the external subset of the Document Type Definition, if it exists
- [public identifier]
  The public identifier of the external subset of the Document Type Definition, if one exists, normalized as described in Chapter 4.2.2 of the XML 1.0 Recommendation
- [children]
  An ordered list of any processing instruction information items representing processing instructions in the original document
- [parent]
  The document information item

Note the major difference between XPath and the XML Information Set with respect to this information item. In XPath there is no representation of the Document Type Declaration.

Unparsed Entity Information Item

There is an unparsed entity information item in the information set for each unparsed general entity declared in the Document Type Definition.

An unparsed entity information item has six properties:

- [name]
  The name of the entity
- [system identifier]
  The system identifier of the entity
- [public identifier]
  The public identifier of the entity normalized as described in Chapter 4.2.2 of the XML 1.0 Recommendation
- [declaration base URI]
  The base URI relative to which the system identifier should be resolved
- [notation name]
  The notation name associated with the entity

- [notation]
  The notation information item named by the notation name

Notation Information Item

There is a notation information item for each notation declared in the Document Type Definition.

A notation information item has four properties:

- [name]
  The name of the notation
- [system identifier]
  The system identifier of the notation
- [public identifier]
  The public identifier of the notation as normalized in Chapter 4.2.2 of the XML 1.0 Recommendation
- [declaration base URI]
  The base URI relative to which the system identifier should be resolved

Namespace Information Item

There is a namespace information item for each namespace declaration that is in scope on each element in the original document.

A namespace information item has two properties:

- [prefix]
  The prefix whose binding this namespace information item describes. The prefix is the string that follows "xmlns:" in the corresponding namespace declaration.
- [namespace name]
  The namespace name to which the prefix is bound.

Perspective

In this chapter I presented an overview of the model followed by XPath 1.0, which forms the basis of the use of XPath described throughout this book.

Increasingly, the use of XML is being carried out using a mixture of technologies. So although not immediately relevant for the use of XPath today, an awareness of the differences in approach in the data models of the DOM and, more particularly, of the XML Information Set is useful knowledge as XPath and the context in which it is applied evolve.

As I read the XML Information Set Candidate Recommendation, it reminded me of XML as it was two years ago, rather than the more data-centric XML of today. In addition, while the XML Information Set will be important, it too will be undergoing ongoing

development as it is updated to recognize the existence of XSD Schema and the pivotal role which XSD Schema seems likely to play in W3C's vision for the future of XML.

XPath will need to adapt and be updated to correspond with those potentially far-reaching changes. An early indication of the direction that this process may take can be seen in the first Working Draft of the XPath 2.0 Requirements document (located at www.w3.org/TR/xpath20req).

XPath 1.0 will not be outdated in the immediate future. I expect the development of XPath 2.0 to be a relatively slow process because of the interdependencies with the XML Query Language, XQuery, with the likely developments in XSLT 2.0 and the need to align more closely with the DOM. I will discuss XPath's likely future in more detail in Chapter 14, "XPath and XQuery."

Looking Ahead

For now let's move on to examine more closely the four syntax forms available in XPath 1.0.

The Four XPath Syntaxes

In this chapter we will look in more detail at the four types of syntax in the XPath Recommendation.

The four different syntaxes available for use in XPath are as follows:

- Unabbreviated absolute syntax
- Unabbreviated relative syntax
- Abbreviated absolute syntax
- Abbreviated relative syntax

In practice, much of the XPath you use in real-world code will be based on the abbreviated relative syntax and that syntax will be given emphasis later in this chapter. However, the unabbreviated syntax more clearly brings out the structure of XPath expressions and location paths, particularly with respect to XPath axes. In addition, not all the possible XPath expressions can be expressed in either of the abbreviated syntaxes, so at times you may need to use the unabbreviated syntax, even if you prefer solid practical reasons to use the abbreviated syntax for most of your code.

If you fully understand the unabbreviated syntax then you will have a good grasp of what XPath 1.0 is capable of, since all XPath 1.0 expressions can be written in the unabbreviated syntax. Therefore the two forms of the unabbreviated syntax will be described first.

Remember that XPath has 13 axes, and we are going to look at all of those in this chapter. Some of the axes can only be expressed in location paths when we use the unabbreviated syntax.

Unabbreviated Absolute Syntax

In this section we will explore the unabbreviated absolute syntax form. In this form of XPath syntax the context node is always the root node. This is determined by all location paths beginning with the "/" character, as in

```
/child::Book/child::Chapter/child::Section
```

which would select the element node corresponding to the <Section> element in the following code:

```
<Book>
<Chapter>
<Section>The section content goes here.
</Section
</Chapter>
</Book>
```

Following the root node in the location path is one or more location steps, each of which consists of an axis, a node test, and an optional predicate.

NOTE **There are certain axes that cannot be meaningfully expressed using the unabbreviated absolute syntax. Nothing precedes the root node in document order, nor does it have any siblings, nor any ancestors. Therefore, practically speaking, when the context node is the root node in the unabbreviated absolute syntax, you cannot use (or there is no point in using) the parent axis, the ancestor axis, the following-sibling axis, the preceding-sibling axis, the following axis, the preceding axis, the attribute axis, the namespace axis, or the ancestor-or-self axis.**

We will use the source document shown in Listing 4.1 in several examples that follow. Let's look at how the location steps work in this syntax form by applying a range of location paths to the following XML source document.

```
<?xml version='1.0'?>
<Document>
 <Sections>
  <Section order="First">
   <Paragraph type="info">
    Here is an initial paragraph in the first section.</Paragraph>
```

Listing 4.1 A Document Represented in XML (Document01.xml).

```
   <Paragraph type="info">
Here is a second paragraph in the first section.
   </Paragraph>
   <Paragraph type="warning">
    Be careful here in this third paragraph.
   </Paragraph>
  </Section>
  <Section order="Second">
   <Paragraph type="info">
    An initial paragraph in the second section.
   </Paragraph>
   <Paragraph type="warning">
    Be careful in this second paragraph in the second section.
   </Paragraph>
  </Section>
  <Section order="Third">
   <Paragraph type="info">
    A lonely paragraph in the third section.
   </Paragraph>
  </Section>
 </Sections>
</Document>
```

Listing 4.1 (*Continued*)

First let's look at the child axis, which can allow us to select element, comment, processing instruction, or text nodes, depending on the precise syntax we use.

The Child Axis

Let's begin our exploration of the child axis by using an XPath location path to output the content of a set of elements in the source document in Listing 4.1.

If we want to display the content of all the <Paragraph> elements in the document, we can use an XPath expression to select the <Paragraph> elements only to be output to the HTML page, as in Listing 4.2.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
```

Listing 4.2 A Stylesheet to Demonstrate the Child Axis. (*continues*)

```
<html>
<body>
<xsl:apply-templates
select="/child::Document/child::Sections/child::Section/child::
  Paragraph"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<h3><xsl:value-of select="self::node()"/></h3><br />
</xsl:template>

</xsl:stylesheet>
```

Listing 4.2 (*Continued*)

The stylesheet produces an HTML document, Document01.html, where the content of each of the <Paragraph> elements is output contained within HTML <h3> tags. From the XPath point of view, the part of the code that does most of the work is the select attribute of the <xsl:apply-templates> element, which contains the following unabbreviated absolute syntax expression:

```
/child::Document/child::Sections/child::Section/child::Paragraph
```

This informs the XSLT processor that the <xsl:template> element to be processed must match the XPath location path given as the value of the select attribute of the <xsl:apply-templates> element.

Let's look a little closer at that location path. There are four location steps in the location path. The first location step indicates that the context node is the root node.

```
/child::Document
```

The child key word indicates that the child axis is to be used. The node test in the location path is "Document," which indicates that we test to see if there are any nodes that represent Document elements found on the child axis, starting at the root node. We know that we are looking for an element node since the principal node type (the default type, if you like) for the child axis is the element node. This location step does not have a predicate, so the node set selected by the first location step is the node that represents the <Document> element, which is also the element root of the document.

The second location step follows the "/" separator between location steps. The location step indicates that we start from the element node that represents the <Document> element and move along the child axis to find any element nodes that represent <Sections> elements.

```
child::Sections
```

This time the location step again selects one node in the node set—the element node representing the <Sections> element in the source document.

The third location step indicates that, starting from the context node (an element node which represents the <Sections> element), we again traverse the child axis and look for element nodes that represent <Section> elements.

```
child::Section
```

On this occasion we find three element nodes that represent <Section> elements.

For each of the element nodes that represent <Section> elements selected by the third location step, we apply the logic of the fourth location step, which again takes us along the child axis to, successively, each of the element nodes that represent the <Paragraph> elements in the source document.

```
child::Paragraph
```

The preceding paragraphs have spelled out step by step that the nodes selected by the <xsl:apply-templates> element are those element nodes that represent <Paragraph> elements, which have <Section> element parents, which, in turn, have a <Sections> element parent, which in turn have a <Document> element parent whose parent is, in turn, the root node. You will likely be glad to know that it takes much more time to describe or read about the detail of the process than it does to write the code to make it all happen.

In other words, when we see the code below, the element nodes which will be processed, given the select attribute of the <xsl:apply-templates> element, are those that represent the <Paragraph> elements just described.

```
<xsl:template match="Paragraph">
```

For each of those element nodes in document order, which represent the <Paragraph> elements, the processor applies the content of the <xsl:template> element. Thus the XSLT processor checks each <Section> element node for how many element nodes representing <Paragraph> element children it has, and for each of those the content of the <xsl:template match=”Paragraph”> is applied. Once all those <Paragraph> elements are processed, the next element node representing a <Section> element is processed, so that all its element node children representing <Paragraph> elements are processed. The same process is repeated for any other element nodes that represent <Section> elements and for each of the nodes that represent their <Paragraph> element children.

What actually happens within the template is that the self axis for the context node is chosen, as expressed in the expression `self::node()`. The result is that the content of the <Paragraph> element is written to the output tree.

As you can see in Figure 4.1 the HTML output simply displays the content of each of the <Paragraph> elements in document order on the HTML page.

The Attribute Axis

Let’s move on and use the attribute axis to display the value of the type attribute on each <Paragraph> element in the document. If we want to output the values of the type attributes of the <Paragraph> elements, then we can do so using the code in Listing 4.3.

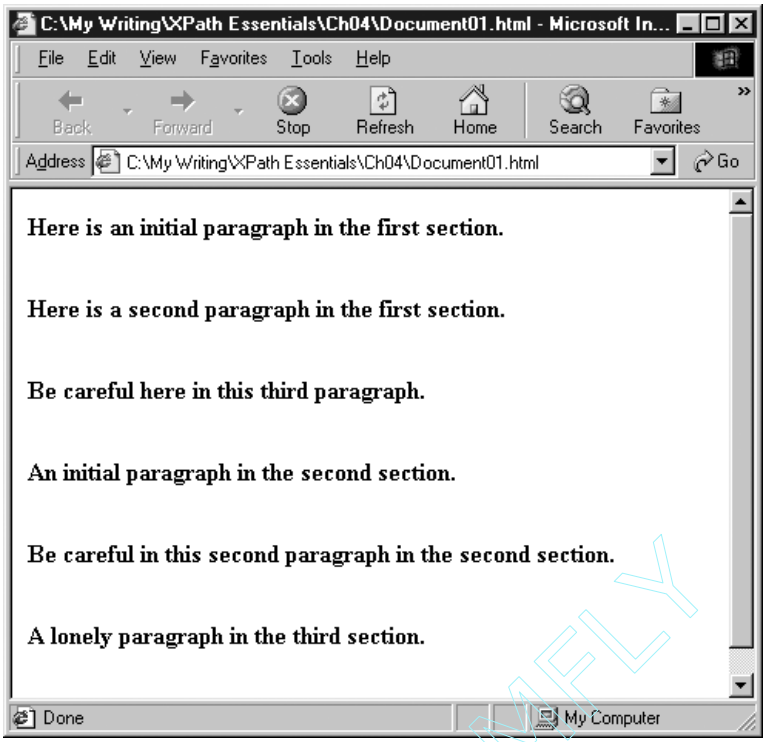

C:\My Writing\XPath Essentials\Ch04\Document01.html - Microsoft In...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Document01.html
Go
Here is an initial paragraph in the first section.
Here is a second paragraph in the first section.
Be careful here in this third paragraph.
An initial paragraph in the second section.
Be careful in this second paragraph in the second section.
A lonely paragraph in the third section.
Done
My Computer

Figure 4.1 Selecting the Content of Element Nodes Representing Paragraph Elements.

TEAMFLY

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="/child::Document/child::Sections/child::Section/child::
  Paragraph"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<h3><xsl:value-of select="attribute::type"/></h3><br />
</xsl:template>

</xsl:stylesheet>
```

Listing 4.3 A Stylesheet to Demonstrate Using the Attribute Axis (Document02.xsl).

The select attributes of the <xsl:apply-templates> element in the main template takes us to the templates that best match the element nodes that represent <Paragraph> elements, as described in detail earlier.

The content of the <xsl:template> element to be applied to each such element node has been changed, with the key part of the code being

```
<xsl:value-of select="attribute::type"/>
```

This indicates that starting from the context node, which is in turn a set of nodes, each of which is an element node representing a <Paragraph> element, we then travel along the attribute axis. If there is an attribute node present that represents a type attribute, then the value of that attribute is included in the output tree.

This time the HTML output, displayed in Figure 4.2, causes the values of the type attribute on each <Paragraph> element to be displayed in document order.

It is straightforward to combine the use of the two axes we have seen in this chapter so far. If we wanted to display the type attribute as a label for the paragraph content, we could modify the content of the <xsl:template match="Paragraph"> to read as in the file Document03.xsl (not shown).

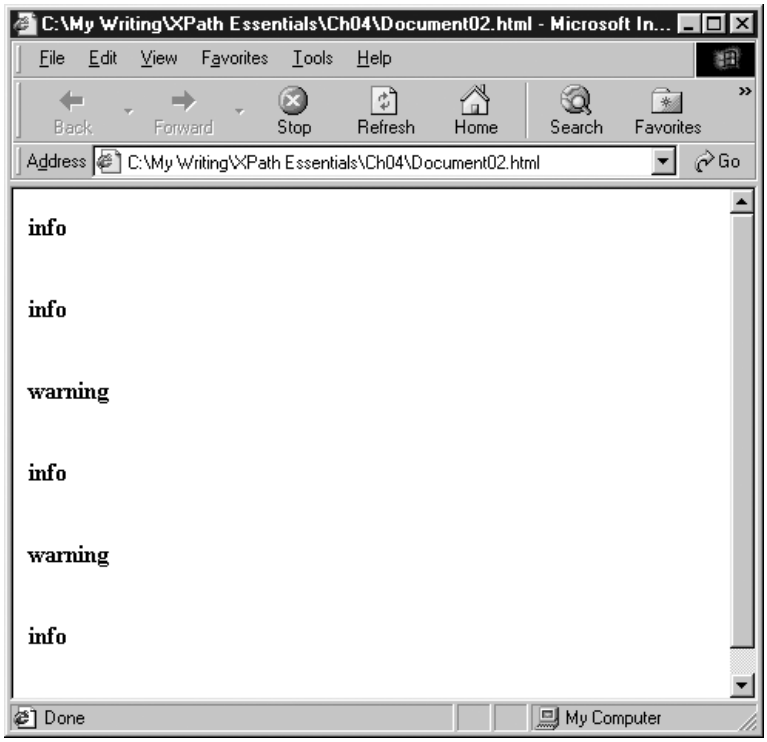


Figure 4.2 Displaying the Content of the Type Attribute of <Paragraph> Elements.

```
<xsl:template match="Paragraph">
<h3><xsl:value-of select="attribute::type"/></h3>
<p><xsl:value-of select="self::node()"/></p>
</xsl:template>
```

This outputs the attribute type as a header label for the content of each <Paragraph> element as shown in Figure 4.3.

We can use location paths that use the attribute axis not only to select for the existence of a particular attribute but also to check that it has a particular value. Thus if we modify the code further to specify the value of the type attribute to be "warning" as in Listing 4.4, we would use the predicate in the value of the select attribute of the <xsl:apply-templates> element to process only <Paragraph> element nodes when they have a type attribute and the type attribute has the value of "warning."

The output of the transformation can be seen in Figure 4.4.

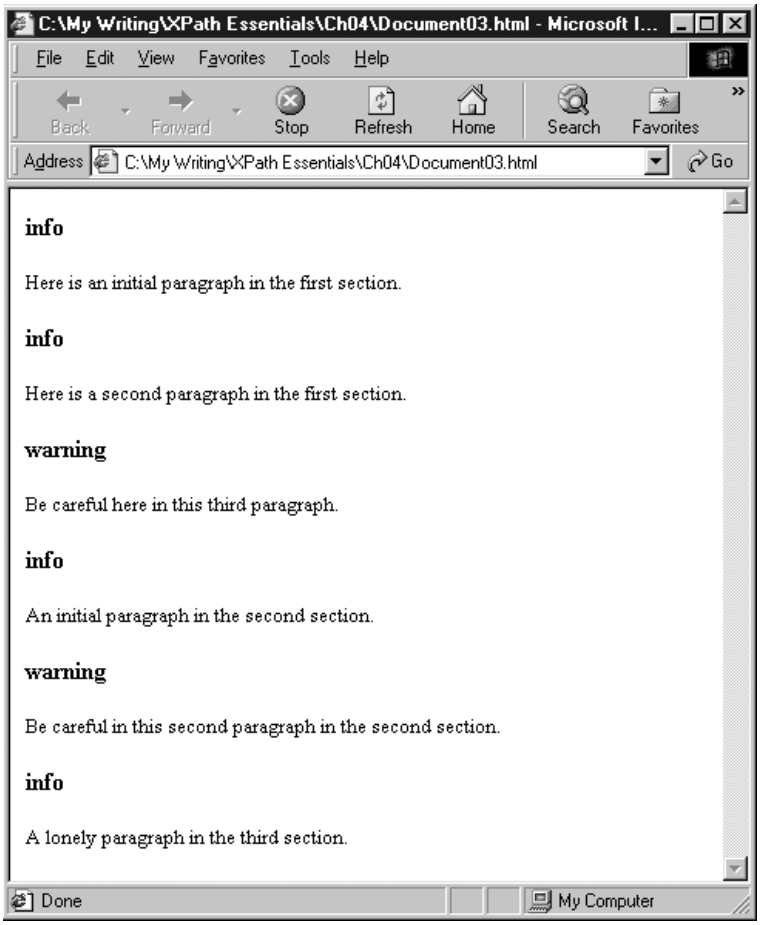

C:\My Writing\XPath Essentials\Ch04\Document03.html - Microsoft I...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Document03.html
Go
info
Here is an initial paragraph in the first section.
info
Here is a second paragraph in the first section.
warning
Be careful here in this third paragraph.
info
An initial paragraph in the second section.
warning
Be careful in this second paragraph in the second section.
info
A lonely paragraph in the third section.
Done
My Computer

Figure 4.3 Displaying the Value of the Type Attribute and the <Paragraph> Element Content.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="/child::Document/child::Sections/child::Section/child::
  Paragraph[attribute::type='warning']"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<h3><xsl:value-of select="attribute::type"/></h3>
<p><xsl:value-of select="self::node()"/></p>
</xsl:template>

</xsl:stylesheet>
```

Listing 4.4 A Stylesheet to Select for Paragraphs of Type "Warning" (Document03a.xsl).

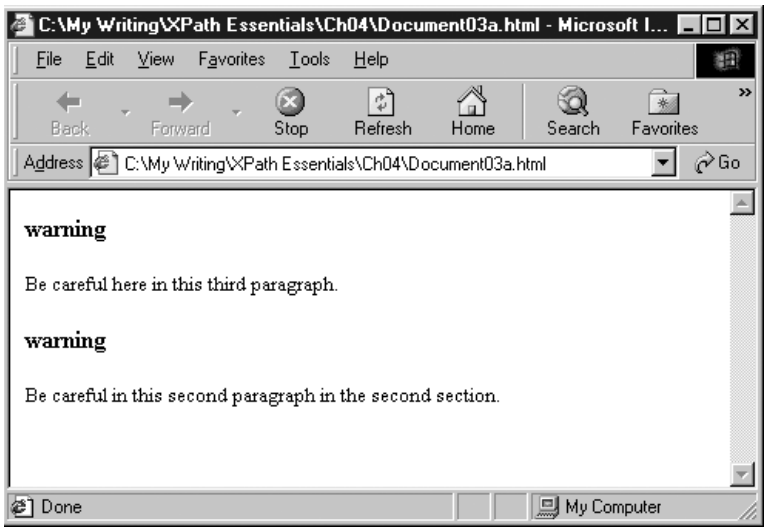

C:\My Writing\XPath Essentials\Ch04\Document03a.html - Microsoft I...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Document03a.html
Go
warning
Be careful here in this third paragraph.
warning
Be careful in this second paragraph in the second section.
Done
My Computer

Figure 4.4 The Result of a Transformation That Displays <paragraph> Elements Only When They Have a Type Attribute Equal to "Warning."

The Descendant Axis

Let's move on to look at how to select nodes in the descendant axis using the unabbreviated absolute syntax.

First we will modify our source document so that it has <Paragraph> elements nested as in the previous version, but also possesses <Paragraph> elements that are present nested in an <Appendix> element, as in Listing 4.5.

First let's use the descendant axis to display the content of all the <Paragraph> elements in the document that are descendants of the root node (that is, all <Paragraph> elements in the document), as in Listing 4.6.

Notice that in the <xsl:apply-templates> element in the main template, the value of the select attribute of the <xsl:apply-templates> element uses the descendant axis:

```
/descendant::Paragraph
```

Therefore, any element node (element nodes are the principal node type for the descendant axis) in the document that also represents a <Paragraph> element will be

```
<?xml version='1.0'?>
<Document>
<Sections>
<Section order="First">
<Paragraph type="info">Here is an initial paragraph in the first
section.</Paragraph>
<Paragraph type="info">Here is a second paragraph in the first
section.</Paragraph>
<Paragraph type="warning">Be careful here in this third
  paragraph.</Paragraph>
</Section>
<Section order="Second">
<Paragraph type="info">An initial paragraph in the second
section.</Paragraph>
<Paragraph type="warning">Be careful in this second paragraph in the
  second
section.</Paragraph>
</Section>
<Section order="Third">
<Paragraph type="info">A lonely paragraph in the third
  section.</Paragraph>
</Section>
</Sections>
<Appendix>
<Paragraph>A first paragraph in the Appendix.</Paragraph>
<Paragraph>A second paragraph in the Appendix.</Paragraph>
</Appendix>
</Document>
```

Listing 4.5 A Document with Sections and Appendices (Document11.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
<xsl:apply-templates select="/descendant::Paragraph"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<h3><xsl:value-of select="."/></h3><br />
</xsl:template>

</xsl:stylesheet>
```

Listing 4.6 A Stylesheet to Demonstrate the Descendant Axis (Document11.xsl).

selected by the code `/descendant::Paragraph`. For our source document this will mean that both the <Paragraph> elements that are children of a <Section> element and the <Paragraph> elements that are children of an <Appendix> element will be chosen.

The output HTML is displayed in Figure 4.5. By comparing the <Paragraph> element content in the source document with Figure 4.5, you will see that the content of both types of <Paragraph> element (those nested within <Section> elements and those nested within <Appendix> elements) are output.

Notice in the source document that the <Paragraph> elements that are children of a <Section> element have a type attribute, whereas the <Paragraph> elements that are children of an <Appendix> element do not. We can make use of that difference to select the first type of <Paragraph> element by adding a predicate to the location path that is using the descendant axis.

To choose only those <Paragraph> elements that possess a type attribute, we can modify the value of the select attribute in the <xsl:apply-templates> element of the main template to the following code in Listing 4.7.

```
<xsl:apply-templates
select="/descendant::Paragraph[attribute::type]"/>
```

The HTML output is identical to that shown in Figure 4.5. The <Paragraph> elements displayed are the same but the method of selecting them has changed. On this occasion they have been chosen because they are descendants of the root node and also possess a type attribute.

If we wanted to select only <Paragraph> elements that are also warning paragraphs as indicated by the value of their type attribute, we could do that by modifying the <xsl:apply-templates> element as follows:

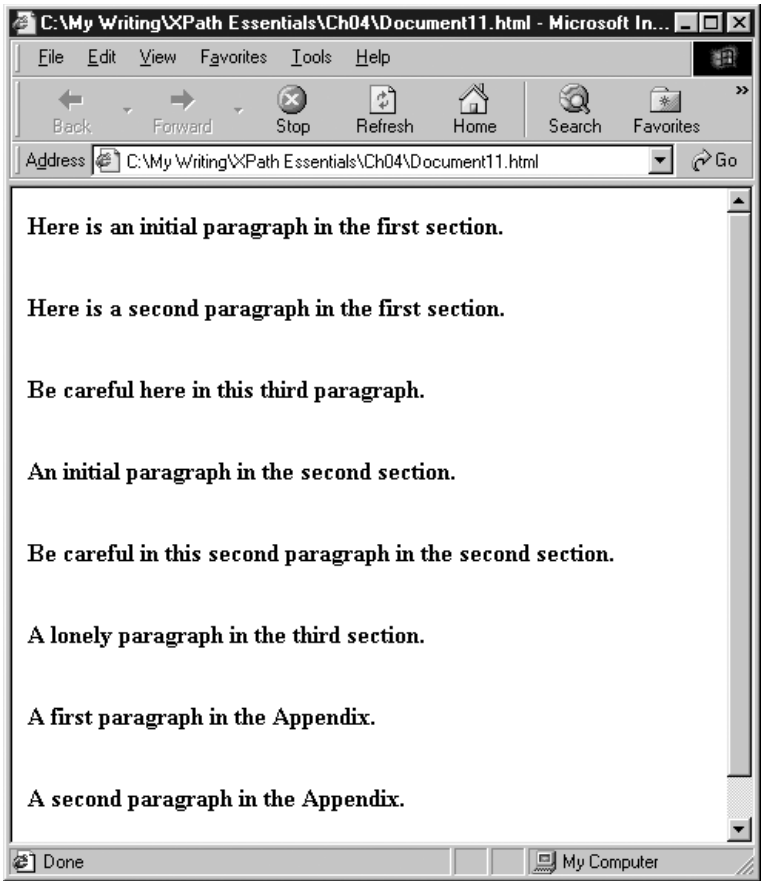

C:\My Writing\XPath Essentials\Ch04\Document11.html - Microsoft In...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Document11.html
Go
Here is an initial paragraph in the first section.
Here is a second paragraph in the first section.
Be careful here in this third paragraph.
An initial paragraph in the second section.
Be careful in this second paragraph in the second section.
A lonely paragraph in the third section.
A first paragraph in the Appendix.
A second paragraph in the Appendix.
Done
My Computer

Figure 4.5 Demonstrating the Descendant Axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="/descendant::Paragraph[attribute::type]"/>
</body>
</html>
```

Listing 4.7 A Stylesheet to Select Paragraphs That Possess a Type Attribute (Document12.xsl).

```
</xsl:template>

<xsl:template match="Paragraph">
<h3><xsl:value-of select="."/></h3><br />
</xsl:template>

</xsl:stylesheet>
```

Listing 4.7 (*Continues*)

```
<xsl:apply-templates
select="/descendant::Paragraph[attribute::type='warning']"/>
```

Be careful that when you add the value of the type attribute to the XPath expression you use a different type of quote than is used for the select attribute. If you have used double quotes around the value of the select attribute, you must use single quotes around the value of the type attribute; otherwise, you will cause an error message when you attempt to carry out the XSLT transformation.

If we make use of the XPath not ( ) function we can reverse the selection so that only <Paragraph> elements that do not possess a type attribute are to be displayed in the HTML output. We modify the <xsl:apply-templates> element so that it looks like this, as in Listing 4.8.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="/descendant::Paragraph[not(attribute::type)]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<h3><xsl:value-of select="."/></h3><br />
</xsl:template>

</xsl:stylesheet>
```

Listing 4.8 A Stylesheet Using the XPath not ( ) Function (Document14.xsl).

```
<xsl:apply-templates
select="/descendant::Paragraph[not(attribute::type)]"/>
```

As you can see in Figure 4.6, this causes only the <Paragraph> elements (which are children of the <Appendix> element) to be displayed, since they are the only <Paragraph> elements in the document that lack a type attribute.

The Parent Axis

The parent axis has no meaning or usefulness with the unabbreviated absolute syntax since the root node does not have a parent node.

The Ancestor Axis

The ancestor axis has no meaning or usefulness with the unabbreviated absolute syntax since the root node does not have any ancestor nodes.

The Following-Sibling Axis

The following-sibling axis has no meaning or usefulness with the unabbreviated absolute syntax since the root node is at the top of the hierarchy and has no siblings.

The Preceding-Sibling Axis

The preceding-sibling axis has no meaning or usefulness with the unabbreviated absolute syntax since the root node is at the top of the hierarchy and has no siblings.

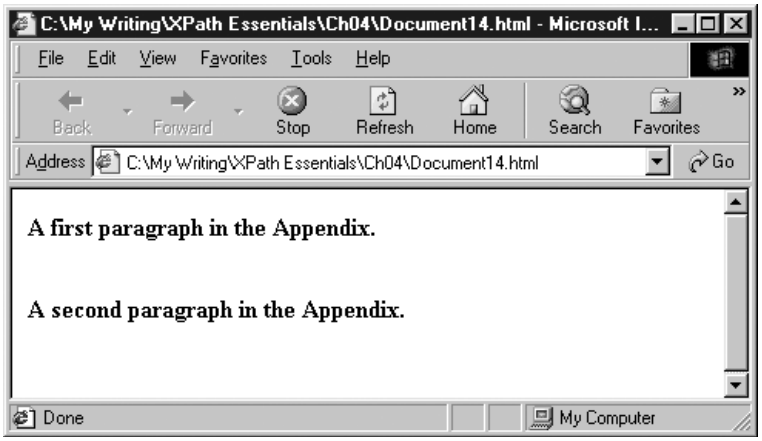

C:\My Writing\XPath Essentials\Ch04\Document14.html - Microsoft I...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Document14.html
Go
A first paragraph in the Appendix.
A second paragraph in the Appendix.
Done
My Computer

Figure 4.6 Selecting Elements for Display Using the not ( ) Function and Presence of a Type Attribute.

The Following Axis

The following axis has no meaning or usefulness with the unabbreviated absolute syntax since the root node is at the top of the hierarchy and all nodes in the tree representing the source document are in the descendant axis; therefore, there is no node in the following axis.

The Preceding Axis

The preceding axis has no meaning or usefulness with the unabbreviated absolute syntax since the root node is at the top of the hierarchy and all nodes in the tree representing the source document are in the descendant axis; therefore, there is no node in the preceding axis.

The Namespace Axis

The XPath specification indicates that it is elements that have associated namespace nodes. Thus, by implication, the root node has no namespace associated with it, not even the default XML namespace.

The Descendant-or-Self Axis

The descendant-or-self axis selects the root node and all nodes that are descendants of the root node (that is, all nodes in the document).

If we apply the stylesheet in Listing 4.9 to our sample source document, you will see that all the element nodes that represent all the elements in the document are output.

If you compare the output of the stylesheet in Figure 4.7 with the source document in Listing 4.5, you will see that the name of each element node has been displayed.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Demonstrating the descendant-or-self axis</title>
</head>
<body>
<h3>Using the descendant-or-self axis.</h3>
<xsl:apply-templates select="/descendant-or-self::*"/>
</body>
</html>
```

Listing 4.9 A Stylesheet to Display All Descendants of the Root Node (Document21.xsl). (*continutes*)

```
</xsl:template>

<xsl:template match="Document | Sections | Section | Paragraph |
Appendix">
<p>There is a &lt;<xsl:value-of select="name()"/>&gt; element
present.</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.9 (*Continued*)

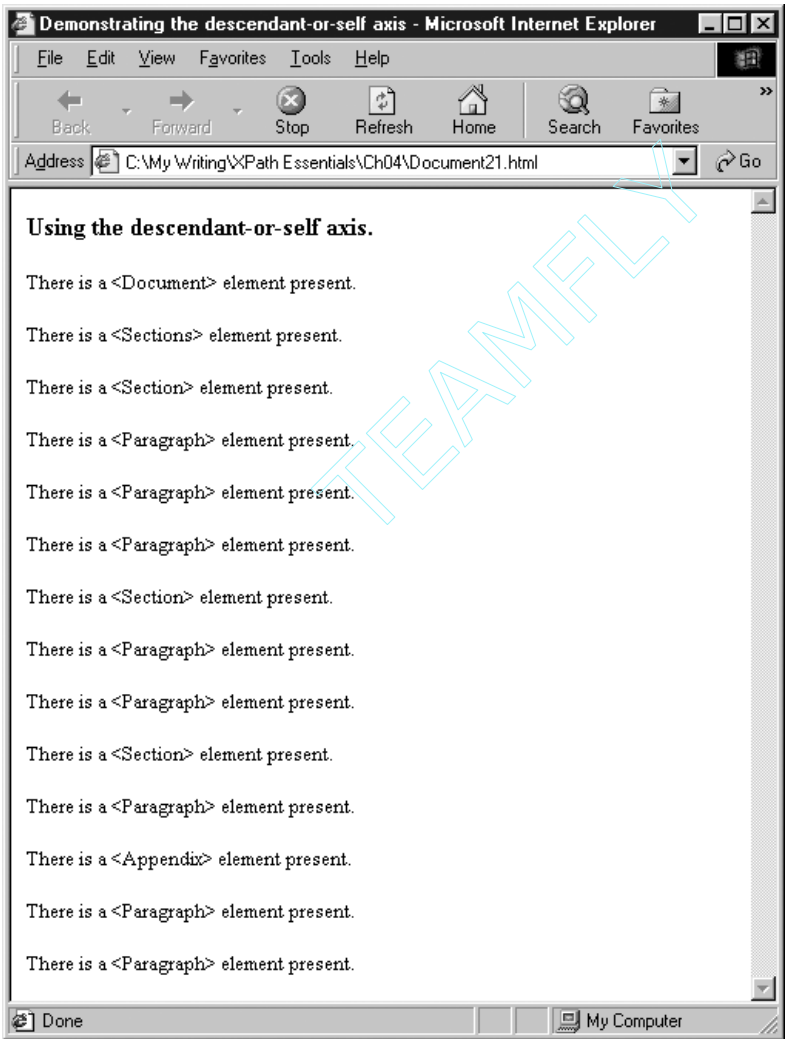


Figure 4.7 Demonstration of the Descendant-or-Self Axis.

The Ancestor-or-Self Axis

The ancestor axis has little meaning or usefulness with the unabbreviated absolute syntax since the root node does not have any ancestor nodes. Thus the ancestor-or-self axis selects only the root node.

The Self Axis

The self axis returns a node set with only one member—the root node itself.

Unabbreviated Relative Syntax

The unabbreviated relative syntax is very similar to the unabbreviated absolute syntax, except that any node may be the context node.

NOTE The Unabbreviated Relative Syntax is the only XPath syntax form in which all 13 axes can be meaningfully expressed or used.

First let's create a source XML document that has sufficient levels of structure to allow us to explore all 13 axes (see Listing 4.10).

```
<?xml version='1.0'?>
<bk:Book xmlns:bk="http://www.xmml.com/Books">
<bk:TableOfContents>
<bk:ChapterTitle number="1">Starting with XML and XSLT</bk:ChapterTitle>
<bk:ChapterTitle number="2">XPath Fundamentals</bk:ChapterTitle>
<bk:ChapterTitle number="3">XPath Data Model</bk:ChapterTitle>
<bk:ChapterTitle number="4">The Four XPath Syntaxes</bk:ChapterTitle>
<bk:Appendix order="A">Online Resources</bk:Appendix>
<bk:Appendix order="B">XPath Glossary</bk:Appendix>
</bk:TableOfContents>
<bk:Introduction>
<bk:Paragraph>
This is the first paragraph of the introduction.
</bk:Paragraph>
<bk:Paragraph>
This is the second paragraph of the introduction.
</bk:Paragraph>
</bk:Introduction>
<bk:Chapters>
<bk:Chapter order="first">
```

Listing 4.10 A Book Represented in XML, Using Namespaces (Book01.xml).

(*continues*)

```
<bk:Paragraph number="1">This is the first paragraph of the first
  chapter.</bk:Paragraph>
<bk:Paragraph number="2">This is the second paragraph of the first
  chapter.</bk:Paragraph>
<bk:Paragraph number="3">This is the third paragraph of the first
  chapter.</bk:Paragraph>
<bk:Paragraph number="4">This is the fourth paragraph of the first
  chapter.</bk:Paragraph>
<bk:Paragraph number="5">This is the fifth paragraph of the first
  chapter.</bk:Paragraph>
<bk:Paragraph number="6">This is the sixth paragraph of the first
  chapter.</bk:Paragraph>
</bk:Chapter>
<bk:Chapter order="second">
<bk:Paragraph number="1">This is the first paragraph of the second
  chapter.</bk:Paragraph>
<bk:Paragraph number="2">This is the second paragraph of the second
  chapter.</bk:Paragraph>
<bk:Paragraph number="3">This is the third paragraph of the second
  chapter.</bk:Paragraph>
<bk:Paragraph number="4">This is the fourth paragraph of the second
  chapter.</bk:Paragraph>
</bk:Chapter>
<bk:Chapter order="third"></bk:Chapter>
<bk:Chapter order="fourth"></bk:Chapter>
</bk:Chapters>
<App:Appendices xmlns:App="http://www.Appendix.com">
<App:Appendix title="Appendix A">Content of Appendix A would go
     here.</App:Appendix>
<App:Appendix title="Appendix B">Content of Appendix B would go
  here.</App:Appendix>
</App:Appendices>
</bk:Book>
```

Listing 4.10 (*Continued*)

Note that the source document includes multiple namespace declarations, which will allow us to explore the namespace axis a little later in this section. The document element, the <bk:Book> element, is declared to belong to the namespace name `http://www.xmml.com/Books` using the following code:

```
<bk:Book xmlns:bk="http://www.xmml.com/Books">
```

All elements nested within the <bk:Book> element will be associated with that namespace URI, except in elements where another namespace is explicitly declared or on the descendants of such elements. In our example document, another namespace is declared on the <App:Appendices> element:

```
<App:Appendices xmlns:App="http://www.Appendix.com">
```

If you don't feel confident handling namespaces, then you may want to take a look at the section on XML namespaces in the latter part of Chapter 1, "Starting with XML and XSLT."

The Child Axis

Let's take a look at using the child axis in the unabbreviated relative syntax, using the stylesheet in Listing 4.11.

Let's take a closer look at the <xsl:apply-templates> element in the main template.

```
<xsl:apply-templates select="child::bk:Book/child::bk:TableOfContents/
  child::bk:ChapterTitle"/>
```

The value of the select attribute is, as before, an XPath location path expressed in the unabbreviated syntax, but this time using the unabbreviated relative syntax. There are three location steps. The first location step starts from the context node, which, in this case, is the root node.

```
child::bk:Book
```

It is the root node, not because there is a leading "/" character as there was in the unabbreviated absolute syntax, but because it is contained in the <xsl:template> element with match attribute equal to "/". In that template the context node is the root node.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
         xmlns:bk="http://www.xmml.com/Books"
         xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="child::bk:Book/child::bk:TableOfContents/child::bk:
  ChapterTitle"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:ChapterTitle">
<h3>Chapter <xsl:value-of select="attribute::number"/>: <xsl:value-of
  select="self::node()"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.11 A Stylesheet to Demonstrate the Child Axis (Book01.xsl).

From the context node we travel the child axis. The node test is "bk:Book". Notice that the <bk:Book> element has a namespace prefix of "bk" separated by a colon from the local part, "Book". Notice also that the "child" key word is followed by a double colon indicating that child is the axis, and that the single colon character is a separator within the QName for the bk:Book element.

The context node is then the element node representing the <bk:Book> element. With that as context node we take the second location step, which selects the element node representing the <TableOfContents> element.

```
child::bk:TableOfContents
```

That becomes the context node for the third location step, which selects a node set containing all the element nodes that represent <ChapterTitle> elements.

```
child::bk:ChapterTitle
```

Therefore, the templates that are instantiated as a result of the following code are those where there is a node representing a <bk:ChapterTitle> element, which is the child of a <bk:TableOfContents> element, which is a child of the <bk:Book> element, which is a child of the root node.

```
<xsl:apply-templates select="child::bk:Book/child::bk:TableOfContents/
  child::bk:ChapterTitle"/>
```

For each of the nodes in that node set, the template in Listing 4.12 is applied, with the output HTML document looking something like Listing 4.13.

```
<xsl:template match="bk:ChapterTitle">
<h3>Chapter <xsl:value-of select="attribute::number"/>: <xsl:value-of
  select="self::node()"/></h3>
</xsl:template>
```

Listing 4.12 produces a display on screen similar to that shown in Figure 4.8.

Of course, the unabbreviated relative syntax can be applied relative to any context node, not just the root node as shown in Listing 4.11. To demonstrate that, we could modify the stylesheet, as in Listing 4.13.

```
<html xmlns:bk="http://www.xmml.com/Books"
      xmlns:App="http://www.Appendix.com">
<body>
<h3>Chapter 1: Starting with XML</h3>
<h3>Chapter 2: XPath Fundamentals</h3>
<h3>Chapter 3: XPath Data Model</h3>
<h3>Chapter 4: The Four XPath Syntaxes</h3>
</body>
</html>
```

Listing 4.12 An HTML Document Representing Four Chapters (Book01.html).

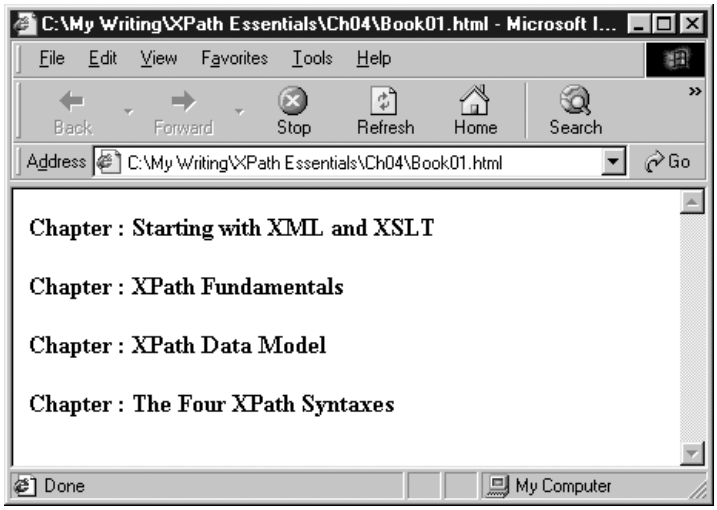

C:\My Writing\XPath Essentials\Ch04\Book01.html - Microsoft I...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch04\Book01.html
Go
Chapter : Starting with XML and XSLT
Chapter : XPath Fundamentals
Chapter : XPath Data Model
Chapter : The Four XPath Syntaxes
Done
My Computer

Figure 4.8 Using the Child Axis in the Unabbreviated Relative Syntax.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="child::bk:Book"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Book">
<p>We briefly visit the node representing the &lt;bk:Book&gt; element.</p>
<p>Then pass on to the &lt;bk:ChapterTitle&gt; element using the select
  attribute of the
&lt;xsl:apply-templates&gt; element.</p>
<xsl:apply-templates
select="child::bk:TableOfContents/child::bk:ChapterTitle"/>
</xsl:template>

<xsl:template match="bk:ChapterTitle">
```

Listing 4.13 A Stylesheet Using Multiple Context Node (Book01b.xsl). (*continues*)

```
<h3>Chapter <xsl:value-of select="attribute::number"/>: <xsl:value-of
select="self::node()"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.13 (*Continued*)

The <xsl:apply-templates> element in the main template passes control to the <xsl:template> element, which matches the <bk:Book> element. The element node representing that element is now the context node. The <xsl:apply-templates> element in that template again alters the context node by means of the location path, which is interpreted relative to the element node representing the <bk:Book> element, not the root node as in the previous example.

```
child::bk:TableOfContents/child::bk:ChapterTitle
```

The HTML output of the stylesheet is displayed in Figure 4.9.

The Attribute Axis

Let's use the attribute axis to display the content of the title attribute of the App:Appendix attribute on the <App:Appendix> element. To do that we will modify our stylesheet, as in Listing 4.14.

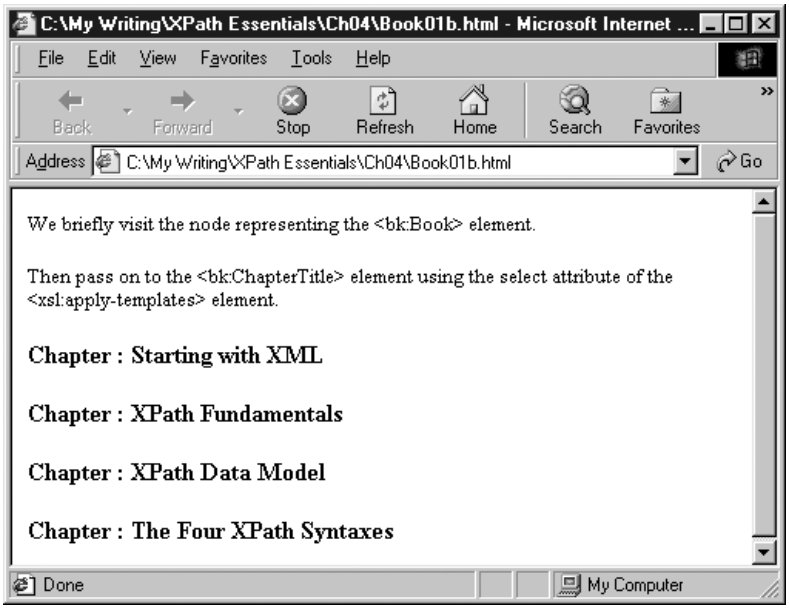

C:\My Writing\XPath Essentials\Ch04\Book01b.html - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book01b.html
Go
We briefly visit the node representing the <bk:Book> element.
Then pass on to the <bk:ChapterTitle> element using the select attribute of the <xsl:apply-templates> element.
Chapter : Starting with XML
Chapter : XPath Fundamentals
Chapter : XPath Data Model
Chapter : The Four XPath Syntaxes
Done
My Computer

Figure 4.9 Using the Unabbreviated Relative Syntax with a Different Context Node.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:bk="http://www.xmml.com/Books"
     xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="child::bk:Book/child::App:Appendices/App:Appendix"/>
</body>
</html>
</xsl:template>

<xsl:template match="App:Appendix">
<h3><xsl:value-of select="attribute::title"/>: <xsl:value-of
select="self::node()"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.14 A Stylesheet to Demonstrate the Attribute Axis (Book02.xsl).

The main changes are within the <xsl:apply-templates> element in the main template and in the template that is instantiated as a result of the <xsl:apply-templates>.

In order to display the value of the title attribute plus the content of the <App:Appendix> elements, we need the appropriate namespace declaration to be present within the start tag of the <xsl:stylesheet> element and to use the namespace prefix used in that namespace declaration.

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:bk="http://www.xmml.com/Books"
     xmlns:App="http://www.Appendix.com">
```

The <xsl:template> element that matches `App:Appendix` in the location path is instantiated and the value of the title attribute as well as the content of the element are output, as shown in Figure 4.10.

The Descendant Axis

Using the descendant axis in the unabbreviated relative syntax is very similar to using it in unabbreviated absolute syntax. If we want to display the content of all the <Paragraph> elements in the source document, we can use the descendant axis to do so, using an XSLT stylesheet, as shown in Listing 4.15.

The HTML output is displayed in a browser similarly to the display in Figure 4.11.

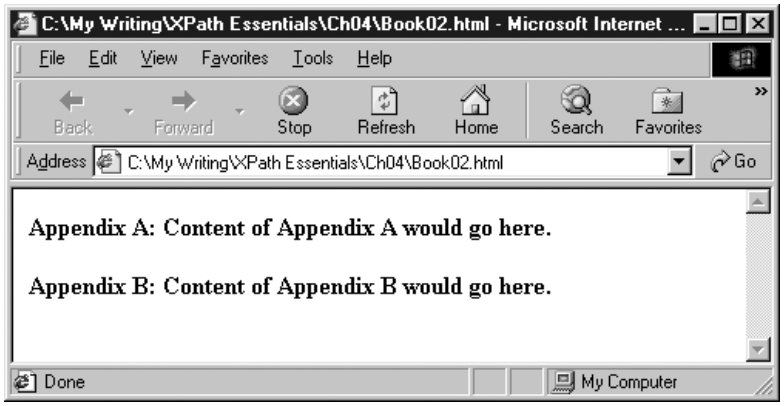

C:\My Writing\XPath Essentials\Ch04\Book02.html - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book02.html
Go
Appendix A: Content of Appendix A would go here.
Appendix B: Content of Appendix B would go here.
Done
My Computer

Figure 4.10 Outputting the Value of the Title Attributes and Content of the <Appendix> Elements.

The problem with using the descendant axis in this fairly indiscriminate way is that everything in the document is displayed, which may not be what you want.

Let's continue to use the descendant axis but use a predicate to refine the <Paragraph> elements that are chosen for display. Let's suppose that we want to display only the content of the <Paragraph> elements that are contained in the second chapter. We can use the stylesheet in Listing 4.16 to select them.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates select="descendant::bk:Paragraph"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="attribute::bk:number"/>: <xsl:value-of
  select="self::node()"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.15 A Stylesheet to Demonstrate the Descendant Axis (Book03.xsl).

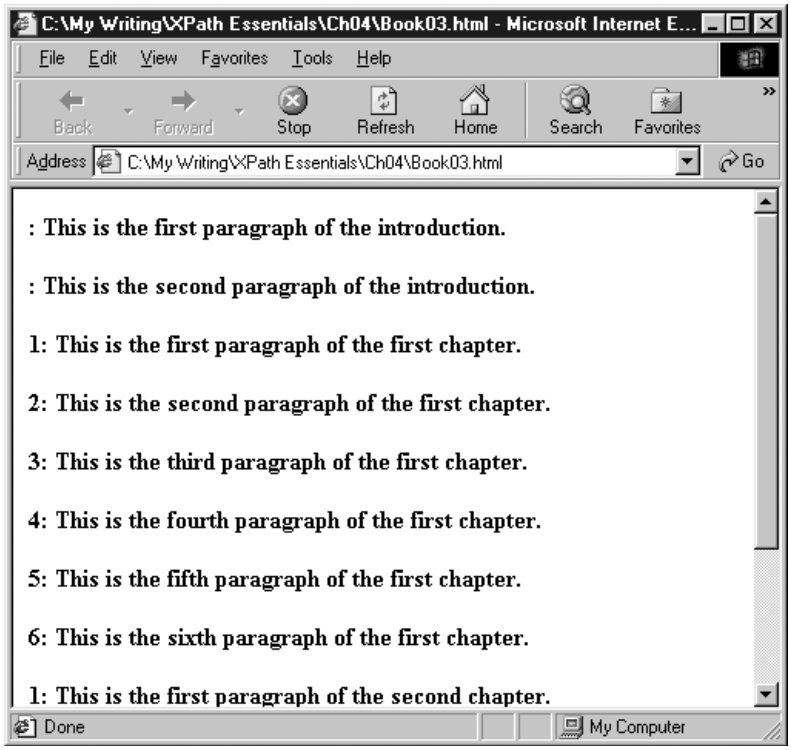

C:\My Writing\XPath Essentials\Ch04\Book03.html - Microsoft Internet E...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book03.html
Go
: This is the first paragraph of the introduction.
: This is the second paragraph of the introduction.
1: This is the first paragraph of the first chapter.
2: This is the second paragraph of the first chapter.
3: This is the third paragraph of the first chapter.
4: This is the fourth paragraph of the first chapter.
5: This is the fifth paragraph of the first chapter.
6: This is the sixth paragraph of the first chapter.
1: This is the first paragraph of the second chapter.
Done
My Computer

Figure 4.11 Using the Descendant Axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:bk="http://www.xmml.com/Books"
    xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="descendant::Chapter[attribute::order='second']/child::bk:
  Paragraph"/>
</body>
</html>
```

Listing 4.16 A Stylesheet to More Selectively Use the Descendant Axis (Book04.xsl).
(*continues*)

```
</xsl:template>

<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="attribute::number"/>: <xsl:value-of
  select="self::node()"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.16 (*Continued*)

If we take a closer look at the select attribute of the <xsl:apply-templates> element in the main template we can dissect it and see how it works.

```
<xsl:apply-templates select="descendant::bk:Chapter[attribute::order=
  'second']/child::bk:Paragraph"/>
```

The first location step looks for element node descendants of the context node (in the main template that is the root node).

```
descendant::bk:Chapter[attribute::order='second']
```

However, the node test selects out of that fairly large number of element nodes only those element nodes that also represent <bk:Chapter> nodes. Out of those nodes that remain selected after the node test has been applied, the predicate selects only those element nodes that not only have an order attribute but also whose bk:attribute node has the value of "second".

```
[attribute::order='second']
```

In other words, the only node that satisfies this first location step is the element node that represents the second <bk:Chapter> element.

The element node that represents that second <bk:Chapter> element then becomes the context node. Thus when the second location step is applied, we start from that single element node as context node and select element nodes in the child axis that represent a <bk:Paragraph> element.

```
child::bk:Paragraph
```

Figure 4.12 shows the output HTML in a browser.

In summary, the location path selects the <Paragraph> elements contained in the second <bk:Chapter> element.

The Ancestor Axis

We were not able to explore the use of the ancestor axis in the unabbreviated absolute syntax since the root node, which is always the context node for that syntax, has no an-

TEAMFLY

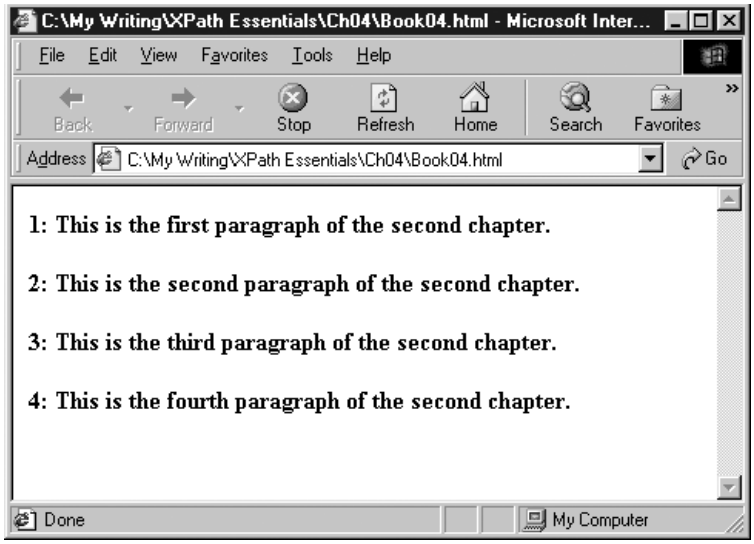

C:\My Writing\XPath Essentials\Ch04\Book04.html - Microsoft Inter...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book04.html
Go
1: This is the first paragraph of the second chapter.
2: This is the second paragraph of the second chapter.
3: This is the third paragraph of the second chapter.
4: This is the fourth paragraph of the second chapter.
Done
My Computer

Figure 4.12 Using the Value of the order Attribute to Filter Output.

cestor. However, when we come to the unabbreviated relative syntax, we can make use of the ancestor syntax if we wish to. The XSLT stylesheet in Listing 4.17 shows an example of how we can do this.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:bk="http://www.xmml.com/Books"
    xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="/child::bk:Book/child::App:Appendices/child::App:Appendix
  [attribute::title='Appendix A']"/>
</body>
</html>
</xsl:template>

<xsl:template match="App:Appendix">
<h3><xsl:value-of select="attribute::title"/>: <xsl:value-of
  select="self::node()"/></h3>
```

Listing 4.17 A Stylesheet to Demonstrate the Ancestor Axis (Book05.xsl).

(*continues*)

```
<h3><xsl:value-of
select="ancestor::node()/child::App:Appendix/attribute::title[self::
  node()='Appendix B']"/>: <xsl:value-of select="ancestor::
  node()/child::App:Appendix[attribute::title='Appendix B']"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.17 (*Continued*)

First we need to take a close look at the select attribute of the <xsl:apply-templates> element in the main template to determine what the context node is.

```
<xsl:apply-templates select="/child::bk:Book/child::App:Appendices/
  child::App:Appendix[attribute::title='Appendix A']"/>
```

In this example, I have used an absolute location path. The first location step starts with the root node as context node and selects element nodes that represent a <bk:Book> element.

```
/child::bk:Book
```

There is one node in the selected node-set. With that node as context node the second location step is applied.

```
child::App:Appendices
```

The node set that is selected following that location step is the element node representing the <Appendices> element. We then apply the third location step.

```
child::App:Appendix[attribute::title='Appendix A']
```

The axis is the child axis and the nodes chosen as a result of the node test are all element nodes that represent <App:Appendix> elements. However, the predicate of the location step refines that node set (which contained two nodes) so that only the element node that has an attribute node with value "Appendix A" remains.

Thus our context node, when we instantiate the following template, is the node that represents the <App:Appendix> element for Appendix A.

```
<xsl:template match="App:Appendix">
<h3><xsl:value-of select="attribute::title"/>: <xsl:value-of
  select="self::node()"/></h3>
<h3><xsl:value-of select="ancestor::node()/child::App:Appendix/
  attribute::title[self::node()='Appendix B']"/>: <xsl:value-of
select="ancestor::node()/child::App:Appendix[attribute::title='Appendix
  B']"/></h3>
</xsl:template>
```

As you can see, the ancestor axis features twice in the above template.

```
ancestor::node()/child::App:Appendix/attribute::title[self::node()
  ='Appendix B']
```

The first location path begins from the element node for the <App:Appendix> element and then searches through the node set of its parent node, that parent's parent, and so on for any of those ancestor nodes that has, as a child node, an element node that represents an <App:Appendix> node that has a title attribute with the value of "Appendix B". Note that when selecting the attribute node to represent the title node, the predicate is what stops the title attribute for the <App:Appendix> element with the title attribute of "Appendix A" from being displayed again.

```
[self::node()='Appendix B']
```

Since this location path is the value of a select attribute in an <xsl:value-of> element, then the value of the attribute is displayed.

The second location path that uses the ancestor axis differs in that it is the element node for the <App:Appendix> element that is selected.

```
ancestor::node()/child::App:Appendix[attribute::title='Appendix B']
```

The predicate acts to select only such element nodes when they possess a title attribute with the value of "Appendix B".

If you are confused about how the predicates work in these two location paths, then go back over them carefully and compare the syntax. In the first location path

```
attribute::title
```

is a node test that has a predicate of

```
[self::node()='Appendix B']
```

In the second location path

```
[attribute::title='Appendix B']
```

is a predicate.

The Following-Sibling Axis

Before we look at the syntax for selecting nodes in the following-sibling axis, let's remind ourselves what the following-sibling axis actually is. The following-sibling axis selects all nodes that have the same parent as the context node and that follow the context node in document order.

In the following example, we will set the context node to the element node that represents the first <bk:Paragraph> element in the first <bk:Chapter> element. So first let's look at the XSLT stylesheet in Listing 4.18, which will allow us to use the following-sibling axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="child::bk:Book/child::bk:Chapters/child::bk:Chapter[position()=
  1]/child::bk:Paragraph[position()=1]"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
  chapter:
<xsl:value-of select="following-
sibling::bk:Paragraph/attribute::number"/>. <xsl:value-of
  select="following-sibling::bk:Paragraph"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.18 A Stylesheet to Demonstrate the Following-Sibling Axis (Book06.xsl).

The context node is chosen by the following code in the main template:

```
<xsl:apply-templates select="child::bk:Book/child::bk:Chapters/
  child::bk:Chapter[position()=1]/child::bk:Paragraph[position()=1]"/>
```

Let's look step by step at how we get to the context node. The first location step takes us to the element node representing the <bk:Book> element.

```
child::bk:Book
```

The second location step takes us to the element node representing the <bk:Chapters> element.

```
child::bk:Chapters
```

The axis in the third location step takes us to child elements of the <bk:Chapters> element.

```
child::bk:Chapter[position()=1]
```

The node test `bk:Chapter` specifies that only element nodes representing <bk:Chapter> elements are to be selected. The predicate [position ( )=1] filters the element nodes representing <bk:Chapter> elements so that only the node representing the element node in position 1 (that is, the first node) is selected.

NOTE Remember that in XPath the position of nodes in a node set is numbered from 1, not zero, as in some programming languages.

The final location step starts from the context node selected in the previous location step.

```
child::bk:Paragraph[position()=1]
```

This final location step in the axis and node test selects the element node(s) that are child element nodes of the context node and that also represent <bk:Paragraph> elements. There are six of these. The predicate `[position()=1]` uses the XPath position() function to filter the six nodes to a single node that is in position 1.

Thus the context node to be used in the following <xsl:template> element is the element node representing the first <bk:Paragraph> element in document order in the first <bk:Chapter> element in document order. With that context node, let's examine what the following <xsl:template> element does.

```
<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
  chapter:
<xsl:value-of select="following-
sibling::bk:Paragraph/attribute::number"/>. <xsl:value-of
  select="following-sibling::bk:Paragraph"/></h3>
</xsl:template>
```

The first <xsl:value-of> element uses the XPath ancestor axis to display the value of the order attribute of the parent <bk:Chapter> element (which is present in the ancestor axis).

```
<xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
```

The second <xsl:value-of> element uses the following-sibling axis.

```
<xsl:value-of select="following-sibling::bk:Paragraph/attribute::
  number"/>
```

It selects the first node in document order that is in the following-sibling axis. I have specified that it represents a <bk:Paragraph> element, but it is not necessary to do that here. The final location step specifies that it is the number attribute's value that is to be displayed.

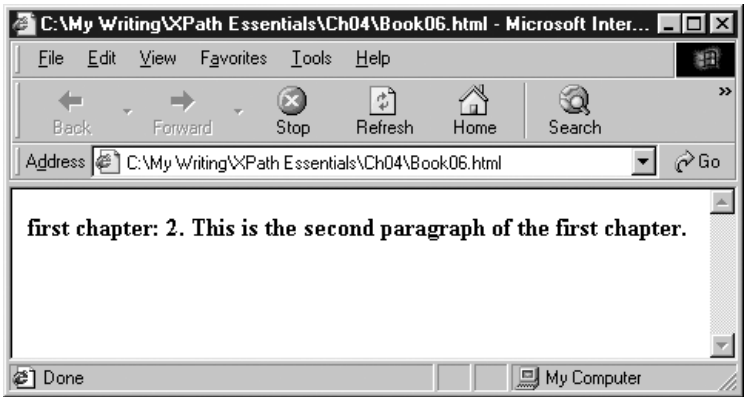

C:\My Writing\XPath Essentials\Ch04\Book06.html - Microsoft Inter...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch04\Book06.html
Go
first chapter: 2. This is the second paragraph of the first chapter.
Done
My Computer

Figure 4.13 Using the Following-Sibling Axis.

The third <xsl:value-of> element simply outputs the text content of the following sibling element node, whose number attribute we have just displayed.

```
<xsl:value-of select="following-sibling::bk:Paragraph"/>
```

The result of running Listing 4.18 on the source XML document is shown in Figure 4.13.

The Preceding-Sibling Axis

Having just seen the following-sibling axis in action, it probably won't surprise you to learn that a definition of the preceding-sibling axis is "The preceding-sibling axis selects all nodes which have the same parent node as the context node and come before the context node in document order." Let's look at an example (see Listing 4.19).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="bk:Book/bk:Chapters/bk:Chapter[position()=1]/bk:Paragraph[
  position()=6]"/>
```

Listing 4.19 A Stylesheet to Demonstrate the Preceding-Sibling Axis (Book07.xsl).

```
</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
  chapter:
<xsl:value-of select="preceding-
sibling::bk:Paragraph[position()=1]/attribute::number"/>. <xsl:
  value-of select="preceding-sibling::bk:Paragraph[position()=1]"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.19 (*Continued*)

If you look at the <xsl:apply-templates> element in the main template you should be able to see that the context node for the following template is the element node representing the sixth <bk:Paragraph> element in the first chapter.

```
<xsl:apply-templates select="bk:Book/bk:Chapters/bk:Chapter
  [position()=1]/bk:Paragraph[position()=6]"/>
```

The first location path in the template is the same as in the example for the following-sibling axis and won't be described further here.

```
<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
  chapter:
<xsl:value-of select="preceding-sibling::bk:Paragraph[position()=1]/
  attribute::number"/>. <xsl:value-of select="preceding-sibling::
  bk:Paragraph[position()=1]"/></h3>
</xsl:template>
```

The second location path contained in the select attribute of the <xsl:value-of> element selects the first preceding-sibling axis *in reverse document order*.

```
<xsl:value-of select="preceding-sibling::bk:Paragraph[position()=1]/
  attribute::number"/>
```

In other words, you count the sibling that is one before the context node as being in position 1, the sibling node before that as being in position 2, and so on.

Similarly, the third location path contained in the following code selects the first preceding element node representing a <bk:Paragraph> element.

```
<xsl:value-of select="preceding-sibling::bk:Paragraph[position()=1]"/>
```

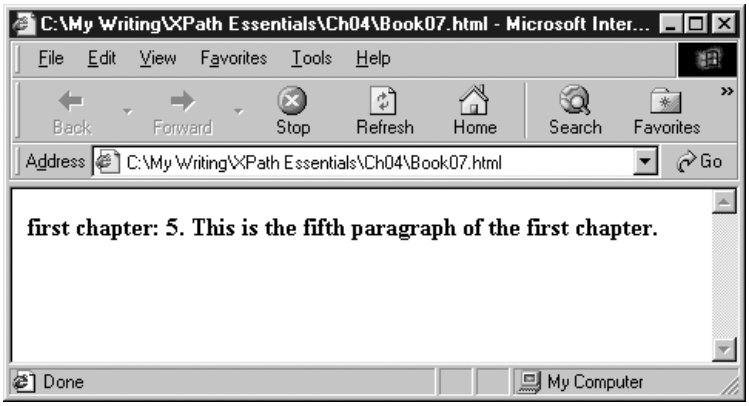

C:\My Writing\XPath Essentials\Ch04\Book07.html - Microsoft Inter...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book07.html
Go
first chapter: 5. This is the fifth paragraph of the first chapter.
Done
My Computer

Figure 4.14 Using the Preceding-Sibling Axis.

Thus the output from our code is the attribute and text of the fifth <bk:Paragraph> element, the first such element encountered starting at the context node of the sixth <bk:Paragraph> element and then proceeding in reverse document order.

The output of the transformation is shown in Figure 4.14.

The Following Axis

In order to explore the following axis, we can use the code in Listing 4.20.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="bk:Book/bk:Chapters/bk:Chapter[position()=1]/bk:Paragraph
  [position()=2]"/>

</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
```

Listing 4.20 A Stylesheet to Demonstrate the Following Axis (Book08.xsl).

```
<h3><xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
  chapter:
<xsl:value-of select="following::bk:Paragraph/attribute::number"/>.
<xsl:value-of select="following::bk:Paragraph"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.20 (*Continued*)

First, let's look at the <xsl:apply-templates> element in the main template:

```
<xsl:apply-templates
select="bk:Book/bk:Chapters/bk:Chapter[position()=1]/bk:Paragraph
  [position()=2]"/>
```

The first two location steps simply select the <bk:Book> element node and the <bk:Chapters> element node, respectively. The third location step selects the <bk:Chapter> element node in position 1.

```
bk:Chapter[position()=1]
```

The fourth location step selects the <bk:Paragraph> element node, which is second within the first chapter.

```
bk:Paragraph[position()=2]
```

Therefore, when the template is instantiated, it is with the <Paragraph> element node in position 2, which is a child node of the Chapter element node in position 1 that is selected.

```
<xsl:template match="bk:Paragraph">
```

Thus, when the template body is processed, the second and third <xsl:value-of> elements cause the <bk:number> attribute node on the following <Paragraph> element node to be output and the content of the following <bk:Paragraph> element node to be output.

We can see in Figure 4.15 that it is the third paragraph in the first chapter whose content is output. This happens because the third paragraph is the first paragraph after the second in document order.

The Preceding Axis

The preceding axis is a reverse axis. We need to remember that as we work through the code in Listing 4.21.

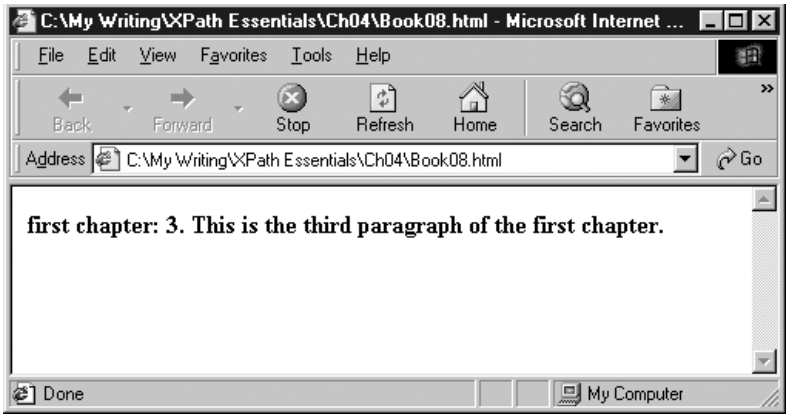

C:\My Writing\XPath Essentials\Ch04\Book08.html - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book08.html
Go
first chapter: 3. This is the third paragraph of the first chapter.
Done
My Computer

Figure 4.15 Using the Following Axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<xsl:apply-templates
select="bk:Book/bk:Chapters/bk:Chapter[position()=1]/bk:Paragraph
  [position()=6]"/>

</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
<h3><xsl:value-of select="."/></h3>
<h3><xsl:value-of select="ancestor::bk:Chapter/attribute::order"/>
  chapter:
<xsl:value-of
select="preceding::bk:Paragraph[position()=1]/attribute::number"/>.
<xsl:value-of select="preceding::bk:Paragraph[position()=1]"/></h3>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.21 A Stylesheet to Demonstrate the Preceding Axis (Book09.xsl).

The <xsl:apply-templates> element in the main template causes the sixth <Paragraph> element node in the first chapter to be selected.

```
<xsl:apply-templates
select="bk:Book/bk:Chapters/bk:Chapter[position()=1]/bk:Paragraph
  [position()=6]"/>
```

Thus when the template

```
<xsl:template match="bk:Paragraph">
```

is instantiated, it is that sixth <Paragraph> element node that is the context node. Thus we must remember that the preceding axis is a reverse axis.

```
<xsl:value-of select="preceding::bk:Paragraph[position()=1]/
  attribute::number"/>
```

The <Paragraph> element node in position 1 is the first element node met going backwards in document order, which is the fifth <Paragraph> element node in the first chapter. We can see the output of the transformation in Figure 4.16.

The Namespace Axis

The source document in Listing 4.10 was designed so we could explore the namespace axis. Listing 4.22 allows us to output the names of the namespace nodes for the <bk:Chapters> and <App:Appendix> elements.

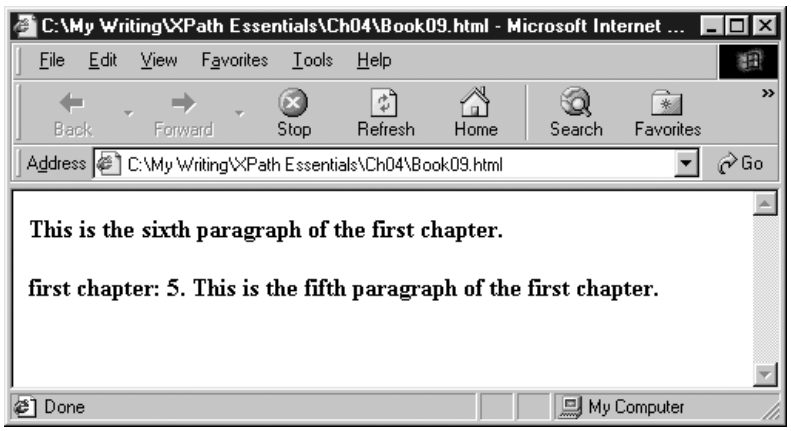

C:\My Writing\XPath Essentials\Ch04\Book09.html - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book09.html
Go
This is the sixth paragraph of the first chapter.
first chapter: 5. This is the fifth paragraph of the first chapter.
Done
My Computer

Figure 4.16 Using the Preceding Axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<h3>The namespace nodes for the &lt;bk:Chapters&gt; element are :</h3>
<xsl:apply-templates
select="bk:Book/bk:Chapters"/>
<h3>The namespace nodes for the &lt;App:Appendices&gt; element are :</h3>
<xsl:apply-templates
select="bk:Book/App:Appendices"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Chapters | App:Appendices">
<xsl:for-each select="namespace::*">
<p>The namespace node name is <xsl:value-of select="name()"/> and is
  associated with the
<xsl:value-of select="."/> namespace.</p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.22 A Stylesheet to Demonstrate Use of the Namespace Axis (Book10.xsl).

First let's look at the output (see Figure 4.17) and then walk through the code to examine why we obtained that output.

The main template contains two <xsl:apply-templates> elements, so let's take those one at a time. Since the template processes the <bk:Chapters> element and the <App:Appendices> element in the same way, we can take them together for at least part of this discussion.

```
<xsl:template match="bk:Chapters | App:Appendices">
```

The <xsl:for-each> element selects, using the location path `namespace::*`, the namespace axis of the respective element nodes. Within the descriptive text, the first <xsl:value-of> element outputs the name of the namespace node. Remember that the name of a namespace node is the local part of the element node which is its parent. Thus for the <bk:Chapters> node we see the output of xml (the local part of the default XML namespace) and of bk. Corresponding to each of the namespace node names, we also output the namespace URI for each.

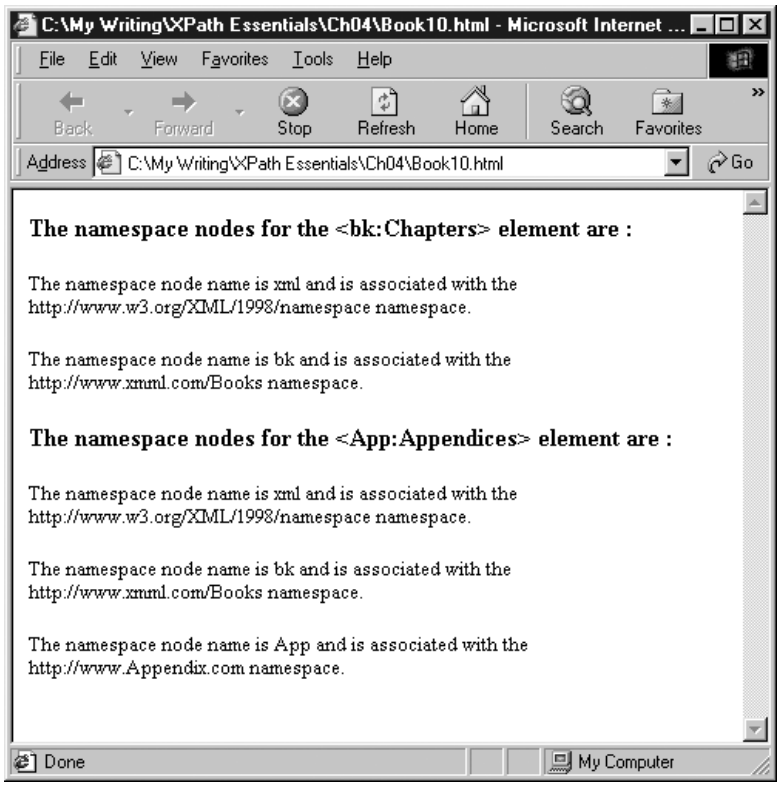

C:\My Writing\XPath Essentials\Ch04\Book10.html - Microsoft Internet ...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book10.html
Go
The namespace nodes for the <bk:Chapters> element are :
The namespace node name is xml and is associated with the http://www.w3.org/XML/1998/namespace namespace.
The namespace node name is bk and is associated with the http://www.xmml.com/Books namespace.
The namespace nodes for the <App:Appendices> element are :
The namespace node name is xml and is associated with the http://www.w3.org/XML/1998/namespace namespace.
The namespace node name is bk and is associated with the http://www.xmml.com/Books namespace.
The namespace node name is App and is associated with the http://www.Appendix.com namespace.
Done
My Computer

Figure 4.17 Using the Namespace Axis.

When we come to examine the output for the <App:Appendices> element, we output the default XML namespace name and URI, as you would probably expect. The App name and the www.Appendix.com namespace may also seem straightforward to you. The output of the bk name and the www.xmml.com/Books namespace URI may have been a little unexpected. The reason for its being there is that the <App:Appendix> element is a child element of the <bk:Book> element; namespaces which are in scope for the parent <bk:Book> element are also in scope for the <App:Appendix> element. Thus, there is namespace node associated with the element node that represents the <App:Appendix> element.

Perhaps you are beginning to think that in the XPath representation of a sizable XML document with several namespaces, there could be many namespace nodes. You'd be correct.

The Descendant-or-Self Axis

The descendant-or-self axis can be a little tricky to use. For example, take a look at the code in Listing 4.23 and try to work out what node(s) will be selected and what will be displayed on screen.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<h3>The name of the node(s) in the descendant-or-self axis are:
<xsl:apply-templates select="descendant-or-self::bk:
  *[position()=3]"/></h3>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Chapter">
<p><xsl:value-of select="name()"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.23 A Stylesheet to Demonstrate the Descendant-or-Self Axis (Book11.xsl).

Let's look at the display of the HTML output by the transformation (see Figure 4.18).

Why has the content of one of the <bk:ChapterTitle> elements in the source document been output? To work that out, let's refresh our memories about the early part of Document01.xml, Listing 4.10:

```
<?xml version='1.0'?>
<bk:Book xmlns:bk="http://www.xmml.com/Books">
<bk:TableOfContents>
<bk:ChapterTitle bk:number="1">Starting with XML and
XSLT</bk:ChapterTitle>
<bk:ChapterTitle bk:number="2">XPath Fundamentals</bk:ChapterTitle>
<bk:ChapterTitle bk:number="3">XPath Data Model</bk:ChapterTitle>
<bk:ChapterTitle bk:number="4">The Four XPath Syntaxes</bk:ChapterTitle>
```

First, let's determine what is selected. The <xsl:apply-templates> element in the main template looks like this:

```
<xsl:apply-templates select="descendant-or-self::bk:*[position()=3]"/
  ></h3>
```

Since we are in the main template, the root node is the context node. The descendant-or-self axis is a forward axis. The element node in position 1 is the element node representing the <bk:Book> element. It is in the node-set since it is a descendant of the root node. The element node in position 2 is the element node representing the

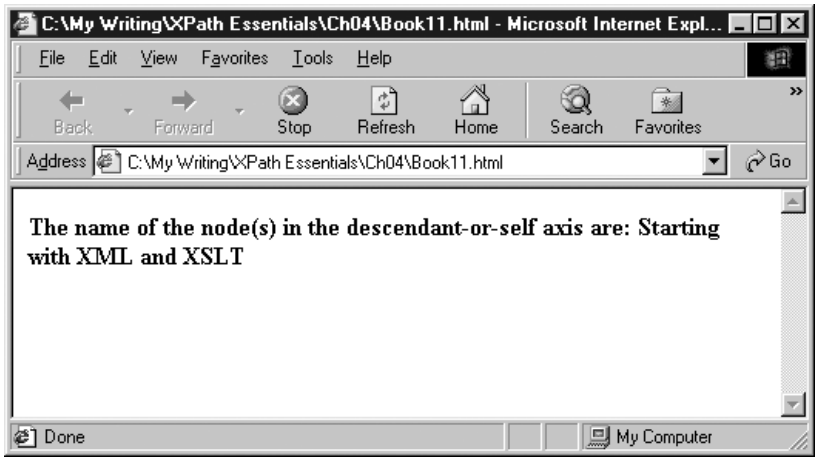

C:\My Writing\XPath Essentials\Ch04\Book11.html - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book11.html
Go
The name of the node(s) in the descendant-or-self axis are: Starting with XML and XSLT
Done
My Computer

Figure 4.18 Output of Transformation Using the Descendant-or-Self Axis.

<bk:TableOfContents> element, which is also a descendant of the root node. The element node in position 3 is the element node representing the first <bk:ChapterTitle> element.

But why is the text content of that element node the output of the transformation? What template is going to be instantiated for the element node that represents the first <ChapterTitle> element? The only other user-defined template is the following, and that doesn't match.

```
<xsl:template match="bk:Chapter">
```

Therefore, the XSLT processor falls back on the default processing template, which is to process the children of the node. In this case, the only child node of the element node is a text node, which contains the text "Starting with XML and XSLT". That content of the text node is what is output when the node selected by <xsl:apply-templates> in the main template is instantiated.

Let's look at another descendant-or-self example to see how things work out when there is a name in the node test, as in Listing 4.24.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
```

Listing 4.24 A Stylesheet to Demonstrate the Descendant-or-Self Axis (Book12.xsl).

(*continues*)

```
<html>
<body>
<h3>The name of the node in the descendant-or-self axis is:
<xsl:apply-templates select="descendant-or-self::bk:Chapter
  [position()=3]"/></h3>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Chapter">
<p><xsl:value-of select="name()"/></p>
<p>The value of its order attribute is <xsl:value-of select="@order"
  /x/p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.24 (*Continued*)

The <xsl:apply-templates> element selects for instantiation the element node representing the <bk:Chapter> element in position 3. As you can see in Figure 4.19, the name of the node is bk:Chapter and the value of its order attribute is "third"; therefore, it is the third bk:Chapter node.

The Parent Axis

There is only one node in the parent axis for any node. Only the root node or an element node can be present in the parent axis for any node. In Listing 4.25, we can see an example of using the parent axis.

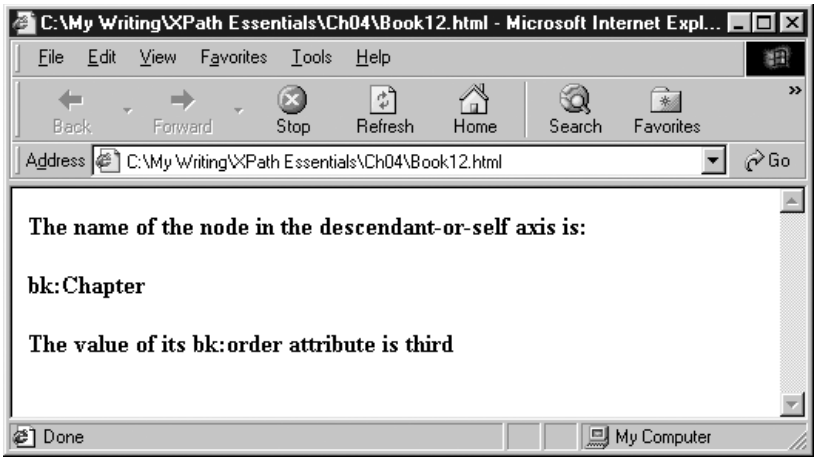

C:\My Writing\XPath Essentials\Ch04\Book12.html - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book12.html
Go
The name of the node in the descendant-or-self axis is:
bk:Chapter
The value of its bk:order attribute is third
Done
My Computer

Figure 4.19 Output of the Transformation using the descendant-or-self Axis from Listing 4.23.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<h3>Using the parent axis</h3>
<xsl:apply-templates select="descendant-or-
self::bk:Chapter[position()=3]"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Chapter">
<p><xsl:value-of select="name()"/></p>
<p>The name of its parent node is <xsl:value-of select="name
  (parent::node())"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.25 A Stylesheet to Demonstrate the Parent Axis (Book13.xsl).

The same element node is selected by <xsl:apply-templates> in the main template, as was selected in the descendant-or-self example we discussed earlier. In the template that matches bk:Chapter, the name of the parent node is output using the following, which uses the name() function to output the name of the parent node:

```
<xsl:value-of select="name(parent::node())"/>
```

In this case it is `bk:Chapters`.

The output of the transformation is shown in Figure 4.20.

The Ancestor-or-Self Axis

The ancestor-or-self axis selects the context node (self) or any of the context node's ancestors, if they match a specified node test.

If we set the context node as the element node representing the third <Paragraph> element in the first chapter, then we can look for an element node representing a <bk:Chapter> element using the code in Listing 4.26.

The context node is as described above. We select its bk:Chapter ancestor node using

```
<xsl:value-of select="ancestor-or-self::bk:Chapter/attribute::bk:order"/>
```

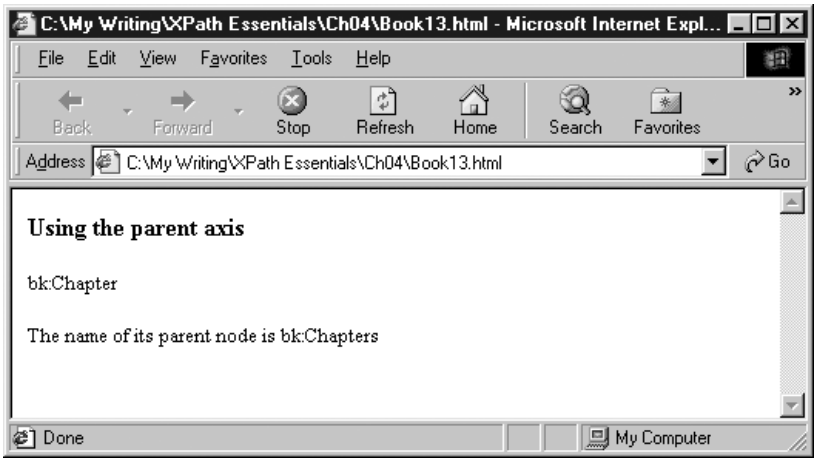

C:\My Writing\XPath Essentials\Ch04\Book13.html - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book13.html
Go
Using the parent axis
bk:Chapter
The name of its parent node is bk:Chapters
Done
My Computer

Figure 4.20 Using the Parent Axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<body>
<h3>Using the parent axis</h3>
<xsl:apply-templates select="descendant-or-self::bk:Chapter
  [position()=1]/bk:Paragraph[position()=3]"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
<p><xsl:value-of select="name()"/></p>
<p>The order attribute of its ancestor &lt;bk:Chapter&gt; node is
<xsl:value-of select="ancestor-or-
self::bk:Chapter/attribute::order"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.26 A Stylesheet to Demonstrate Use of the Ancestor-or-Self Axis (Book14.xsl).

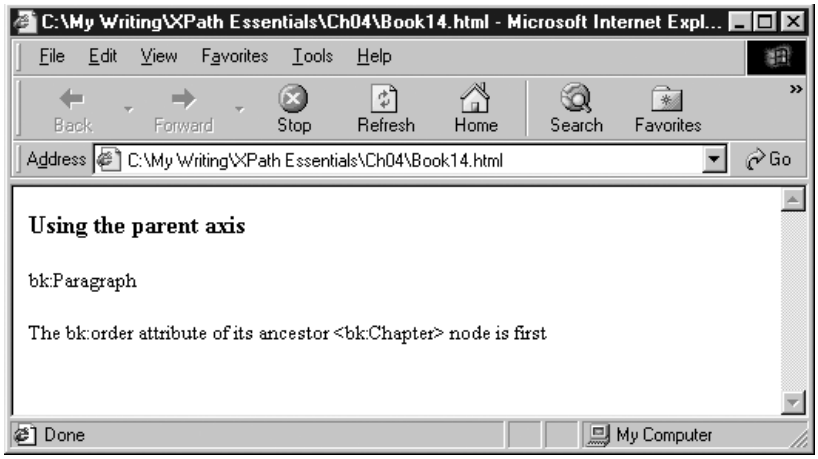

C:\My Writing\XPath Essentials\Ch04\Book14.html - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book14.html
Go
Using the parent axis
bk:Paragraph
The bk:order attribute of its ancestor <bk:Chapter> node is first
Done
My Computer

Figure 4.21 Using the Ancestor-or-Self Axis.

and add a further location step to output the value of the order attribute on the <bk:Chapter> element.

The output of the transformation is shown in Figure 4.21.

The Self Axis

The self axis simply selects the context node. Listing 4.27 shows an example.

The <xsl:apply-templates> element in the main template selects the element node representing the second paragraph in the second chapter.

```
<xsl:template match="bk:Paragraph">
```

The template outputs the name of the node in the self axis using `self::node()`. We can also test for whether the name of the node in the self axis is bk:Chapter and express the result using the boolean ( ) function.

The output of the transformation using the self axis is shown in Figure 4.22.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
```

Listing 4.27 A Stylesheet to Demonstrate the Self Axis (Book15.xsl).

(*continues*)

```
<xsl:template match="/">
<html>
<body>
<h3>Using the parent axis</h3>
<xsl:apply-templates select="descendant-or-
self::bk:Chapter[position()=2]/bk:Paragraph[position()=2]"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:Paragraph">
<p>The name of the self node is <xsl:value-of
select="name(self::node())"/></p>
<p>A test for the self node being a &lt;bk:Chapter&gt; element node is
<xsl:value-of select="boolean(self::bk:Chapter)"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.27 (*Continued*)

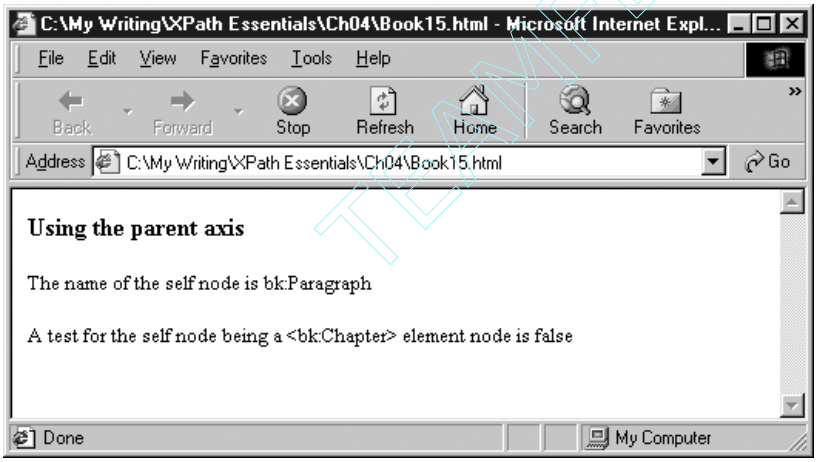

C:\My Writing\XPath Essentials\Ch04\Book15.html - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book15.html
Go
Using the parent axis
The name of the self node is bk:Paragraph
A test for the self node being a <bk:Chapter> element node is false
Done
My Computer

Figure 4.22 Using the Self Axis.

Abbreviated Syntax

XPath has two forms of the abbreviated syntax.

The abbreviated relative syntax is the syntax you are likely to use most frequently. This syntax allows you to express many of the location paths you are likely to need, but there are several parts of XPath that can be expressed in only unabbreviated syntax.

First, we will look at the abbreviated absolute syntax.

Abbreviated Absolute Syntax

The abbreviated absolute syntax uses the root node as the context node by including a "/" character as the first character in each expression.

The Child Axis

The child axis is the *de facto* default axis in XPath; therefore, the name of the child axis may be omitted.

```
/child::Chapter
```

The location path in the unabbreviated absolute syntax may be written in the abbreviated absolute syntax as

```
/Chapter
```

I will continue to use Listing 4.10 as the source document for transformations. Listing 4.28 shows the use of the abbreviated absolute syntax to express the use of the child axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>The child axis in abbreviated absolute syntax</title>
</head>
<body>
<h3>Using the child axis</h3>
<xsl:apply-templates select="/bk:Book/bk:TableOfContents"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:TableOfContents">
<p>The name of the selected child node is <xsl:value-of
select="name(self::node())"/></p>
<p>Its content is <xsl:value-of select="self::node()"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.28 A Stylesheet to Demonstrate the Child Axis (Book20.xsl).

The <xsl:apply-templates> element in the main template starts at the root node as indicated by the initial "/" character.

```
<xsl:apply-templates select="/bk:Book/bk:TableOfContents"/>
```

The next location step is "bk:Book", which is the same as "child::bk:Book" in unabbreviated syntax. The final location step is "bk:TableOfContents", which means the same as "child::bk:TableOfContents" in the unabbreviated syntax.

The output of the transformation is shown in Figure 4.23.

The Attribute Axis

The attribute axis in abbreviated syntax is indicated by the "@" character. Thus `@title` in abbreviated syntax means the same as `attribute::title` in unabbreviated syntax.

The example in Listing 4.29 shows the use of abbreviated absolute syntax to output the value of the bk:number attributes of the <bk:ChapterTitle> elements in the source document.

The template selected by the <xsl:apply-templates> element in the main template uses the absolute location path to output the value of the first number attribute in document order.

```
/bk:Book/bk:TableOfContents/bk:ChapterTitle/@number
```

The Descendant Axis

The descendant axis selects all nodes that are present in the document, since all nodes in the document are descendants of the root node. Listing 4.30 demonstrates the use of the descendant axis in absolute abbreviated syntax.

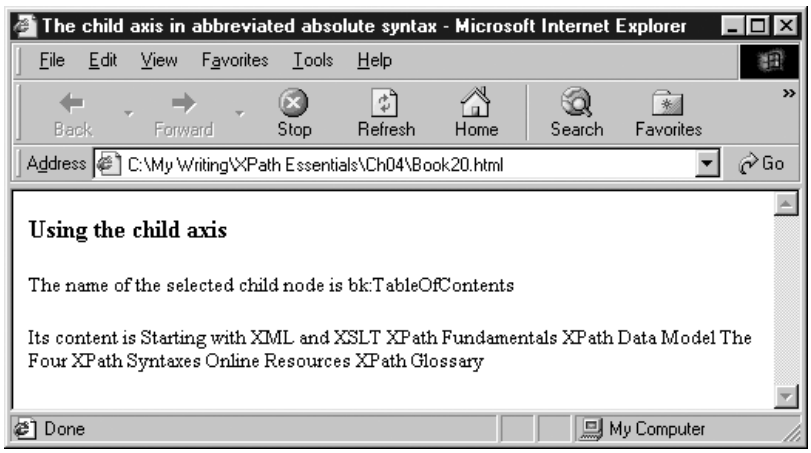

The child axis in abbreviated absolute syntax - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book20.html
Go
Using the child axis
The name of the selected child node is bk:TableOfContents
Its content is Starting with XML and XSLT XPath Fundamentals XPath Data Model The Four XPath Syntaxes Online Resources XPath Glossary
Done
My Computer

Figure 4.23 Using the Child Axis in Abbreviated Absolute Syntax.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>The attribute axis in abbreviated absolute syntax</title>
</head>
<body>
<h3>Using the attribute axis</h3>
<xsl:apply-templates select="/bk:Book/bk:TableOfContents"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:TableOfContents">
<p>The value of the first attribute node is
<xsl:value-of
select="/bk:Book/bk:TableOfContents/bk:ChapterTitle/@number"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.29 A Stylesheet to Demonstrate the Attribute Axis (Book21.xsl).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>The descendant axis in abbreviated absolute syntax</title>
</head>
<body>
```

Listing 4.30 A Stylesheet to Demonstrate the Descendant Axis (Book22.xsl).

(*continues*)

```
<h3>Using the descendant axis</h3>
<xsl:apply-templates select="//bk:ChapterTitle"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:ChapterTitle">
<p>This chapter title is:
<xsl:value-of select="self::node()"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.30 (*Continued*)

The location path used in the <xsl:apply-templates> element in the main template

```
//bk:ChapterTitle
```

is equivalent to the location path in unabbreviated absolute syntax.

```
/descendant::bk:ChapterTitle
```

Strictly speaking, //bk:ChapterTitle is short for /descendant-or-self::node()/child::bk:ChapterTitle, but since the context node is the root node, it is equivalent to the descendant axis when used with a name as the node test in a location step.

The transformation selects all <bk:ChapterTitle> element descendants of the root node; thus all <bk:ChapterTitle> elements in the document are shown in Figure 4.24.

The Parent Axis

The parent axis cannot be meaningfully used with abbreviated absolute syntax since the root node has no parent node.

The Ancestor Axis

The ancestor axis cannot be meaningfully used with abbreviated absolute syntax since the root node has no ancestor node.

The Following-Sibling Axis

The following-sibling axis cannot be meaningfully used with the abbreviated absolute syntax since the root node has no sibling nodes.

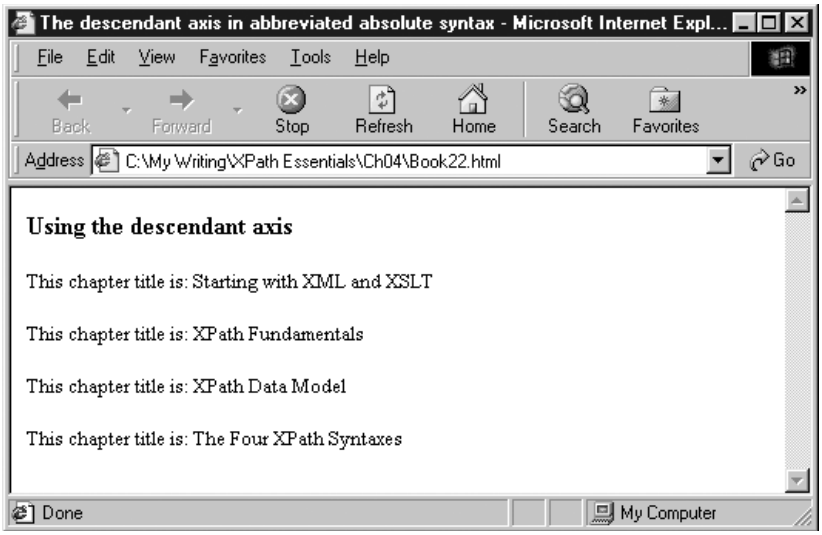

The descendant axis in abbreviated absolute syntax - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book22.html
Go
Using the descendant axis
This chapter title is: Starting with XML and XSLT
This chapter title is: XPath Fundamentals
This chapter title is: XPath Data Model
This chapter title is: The Four XPath Syntaxes
Done
My Computer

Figure 4.24 Using the Descendant Axis in Abbreviated Absolute Syntax.

The Preceding-Sibling Axis

The preceding-sibling axis cannot be meaningfully used with abbreviated absolute syntax since the root node has no sibling nodes.

The Following Axis

There is no syntax to express the following axis in abbreviated absolute syntax.

The Preceding Axis

The preceding axis cannot be meaningfully used with abbreviated absolute syntax since nothing precedes the root node in document order.

The Namespace Axis

There is no syntax to express the namespace axis in the abbreviated absolute syntax.

The Descendant-or-Self Axis

The syntax //Paragraph will select all element nodes in the same document that represent <Paragraph> elements. The root node is not selected since it does not have a name of Paragraph.

The Ancestor-or-Self Axis

The ancestor-or-self axis in the abbreviated absolute syntax cannot return any ancestor nodes since the root node has none.

The Self Axis

The self axis is not particularly useful in the abbreviated absolute syntax since the context node is the root node.

Abbreviated Relative Syntax

Abbreviated relative syntax is the form of XPath you will likely use most frequently. It isn't possible to express all location paths in the abbreviated relative syntax, but you can express the most commonly used axes—the child and attribute axes.

The abbreviated relative syntax uses the same syntax forms as the abbreviated absolute syntax discussed in the previous section, but with the omission of the initial "/" character.

The Child Axis

The example in Listing 4.31 illustrates the use of the child axis using abbreviated relative syntax.

The <xsl:apply-templates> element in the main template selects element nodes that represent <bk:TableOfContents> elements as the context node. Within the template

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>The child axis in abbreviated relative syntax</title>
</head>
<body>
<h3>Using the child axis</h3>
<xsl:apply-templates select="//bk:TableOfContents"/>
</body>
</html>
</xsl:template>
```

Listing 4.31 A Stylesheet to Demonstrate the Child Axis (Book30.xsl).

```
<xsl:template match="bk:TableOfContents">
<p>The child element node(s) of this node are: </p>
<xsl:for-each select="*">
<p><xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.31 (*Continued*)

below, the select attribute of the <xsl:for-each> has the value of "*", which is equivalent to `child::*` in unabbreviated syntax.

```
<xsl:template match="bk:TableOfContents">
```

The context node is the <TableOfContents> element node and it is the child axis relative to that node that the location path in the select attribute of the <xsl:for-each> element evaluates.

The <xsl:for-each> element selects all element children of the <bk:TableOfContents> element. Thus the content of both <bk:ChapterTitle> and <bk:Appendix> elements are output as shown in Figure 4.25.

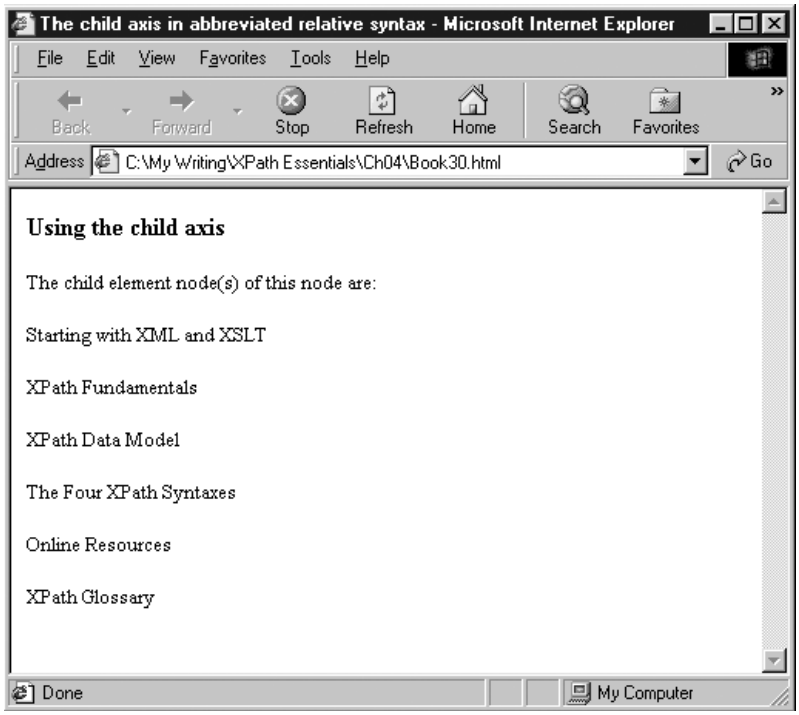

The child axis in abbreviated relative syntax - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address C:\My Writing\XPath Essentials\Ch04\Book30.html
Go
Using the child axis
The child element node(s) of this node are:
Starting with XML and XSLT
XPath Fundamentals
XPath Data Model
The Four XPath Syntaxes
Online Resources
XPath Glossary
Done
My Computer

Figure 4.25 Using the Child Axis in Abbreviated Relative Syntax.

The Attribute Axis

The attribute axis is expressed using an initial "@" character. For example, "@title" would select the attribute node with name of title whose parent is the context node, if such an attribute node exists.

The example in Listing 4.32 demonstrates the use of the attribute axis in abbreviated relative syntax.

The template matches both <bk:ChapterTitle> elements and <bk:Appendix> elements.

```
<xsl:template match="bk:ChapterTitle | bk:Appendix">
```

The select attribute of the <xsl:for-each> element selects all attribute nodes on those elements for which the template is instantiated. The syntax `@*` in abbreviated relative syntax is equivalent to `attribute::*` in unabbreviated relative syntax.

The output of the transformation is shown in Figure 4.26.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>The attribute axis in abbreviated relative syntax</title>
</head>
<body>
<h3>Using the attribute axis</h3>
<p>The attribute node(s) of child nodes of the &lt;TableOfContents&gt;
  element are: </p>
<xsl:apply-templates select="//bk:TableOfContents/*"/>
</body>
</html>
</xsl:template>

<xsl:template match="bk:ChapterTitle | bk:Appendix">
<xsl:for-each select="@*">
<p>On the element with content <xsl:value-of select=".."/> there is an
  attribute named <xsl:value-of select="name()"/>
 whose value is <xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 4.32 A Stylesheet to Demonstrate the Attribute Axis (Book31.xsl).

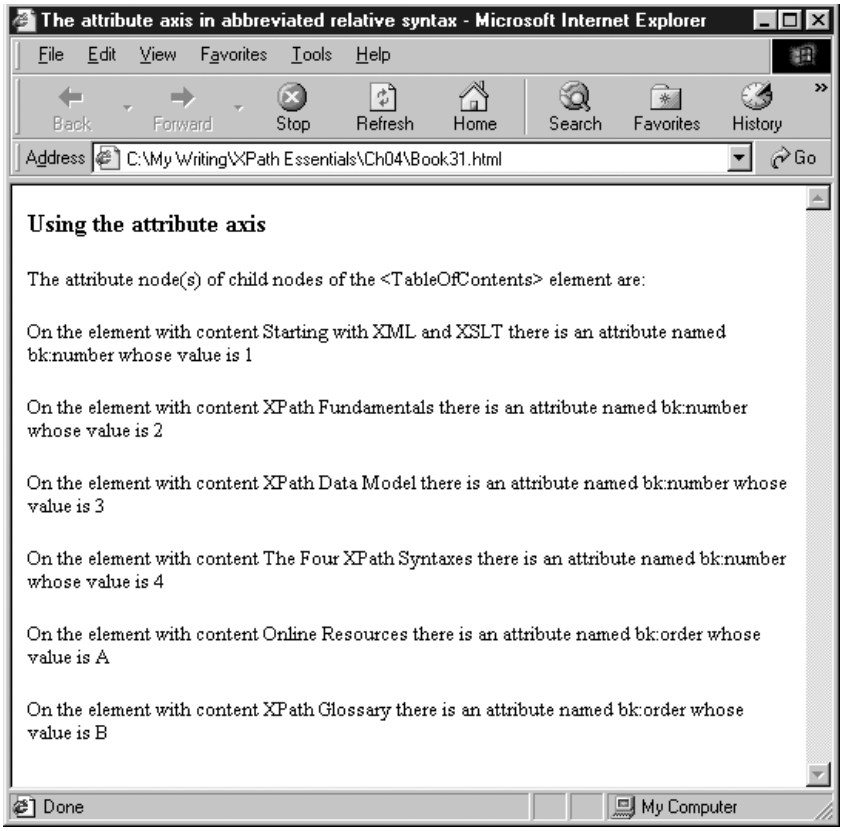

The attribute axis in abbreviated relative syntax - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address C:\My Writing\XPath Essentials\Ch04\Book31.html
Go
Using the attribute axis
The attribute node(s) of child nodes of the <TableOfContents> element are:
On the element with content Starting with XML and XSLT there is an attribute named bk:number whose value is 1
On the element with content XPath Fundamentals there is an attribute named bk:number whose value is 2
On the element with content XPath Data Model there is an attribute named bk:number whose value is 3
On the element with content The Four XPath Syntaxes there is an attribute named bk:number whose value is 4
On the element with content Online Resources there is an attribute named bk:order whose value is A
On the element with content XPath Glossary there is an attribute named bk:order whose value is B
Done
My Computer

Figure 4.26 Using the Attribute Axis in Abbreviated Relative Syntax.

The Descendant Axis

We can use the "//" syntax to select descendants of the context node provided that the accompanying node test excludes the context node.

The example in Listing 4.33 shows the selection of <Paragraph> elements that are descendants of <App:Appendices> elements.

Note the location path below, which means start with the context node using the self axis and select all nodes that are of the principal element type for the descendant axis (that is, the element node).

```
.//*
```

The Parent Axis

The parent axis is simply expressed as "..", which is the equivalent of `parent::node()` in unabbreviated syntax.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:bk="http://www.xmml.com/Books"
                xmlns:App="http://www.Appendix.com">

<xsl:output method="html" indent="yes"/>
<xsl:template match="/">
<html>
<head>
<title>The descendant axis in abbreviated relative syntax</title>
</head>
<body>
<h3>Using the descendant axis</h3>
<p>The descendant node(s) of the &lt;App:Appendices&gt; element
  are: </p>
<xsl:apply-templates select="//App:Appendices"/>
</body>
</html>
</xsl:template>

<xsl:template match="App:Appendices">
<xsl:for-each select=".//*">
<p>Content of a descendant element: <xsl:value-of select="."/> </p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

TEAMFLY

Listing 4.33 A Stylesheet to Demonstrate the Descendant Axis (Book32.xsl).

The Ancestor Axis

The ancestor axis cannot be expressed in abbreviated relative syntax.

The Following-Sibling Axis

The following-sibling axis cannot be expressed in abbreviated relative syntax.

The Preceding-Sibling Axis

The preceding-sibling axis cannot be expressed using abbreviated relative syntax.

The Following Axis

The following axis cannot be expressed in abbreviated relative syntax.

The Preceding Axis

The preceding axis cannot be expressed in abbreviated relative syntax.

The Namespace Axis

The namespace axis cannot be expressed in abbreviated relative syntax.

The Descendant-or-Self Axis

The descendant-or-self axis can be selected using abbreviated relative syntax using the syntax shown in the section for the descendant axis above.

The Ancestor-or-Self Axis

The ancestor-or-self axis cannot be expressed in abbreviated relative syntax.

The Self Axis

The self axis is simply expressed by ".", which is the equivalent abbreviated syntax to `self::node()` in unabbreviated syntax.

Looking Ahead

In this chapter you have seen many examples of how to use XPath to make selections for processing using XSLT. In Chapter 5 we will look in detail at the core function library provided in XPath as well as XSLT functions that can be used together with those XPath functions.

XPath Functions

This chapter will describe, with examples, the functions built into the XML Path Language.

XPath functions operate on (or return) an object of one of four types:

- Node set
- Number
- String
- Boolean

Similarly, the arguments for all XPath functions may be one of the four types listed above.

The Why of XPath Functions

Let's pause for a moment to think about the why of XPath functions. If the purpose of XPath is to address parts of an XML source document, why do we need functions at all?

The answer is fairly simple. Functions provide us with more sophisticated, subtler ways of making selections within the source tree. The use of XPath functions takes us beyond being able to simply select nodes by name.

```
<?xml version='1.0'?>
<AnnualReports>
<AnnualReport Year="1995">Place holder for 1995 report.</AnnualReport>
<AnnualReport Year="1996">Place holder for 1996 report.</AnnualReport>
<AnnualReport Year="1997">Place holder for 1997 report.</AnnualReport>
<AnnualReport Year="1998">Place holder for 1998 report.</AnnualReport>
<AnnualReport Year="1999">Place holder for 1999 report.</AnnualReport>
<AnnualReport Year="2000">Place holder for 2000 report.</AnnualReport>
<AnnualReport Year="2001">Place holder for 2001 report.</AnnualReport>
</AnnualReports>
```

Listing 5.1 A Catalog of Annual Reports Expressed in XML (AnnualReports.xml).

Suppose we have a corporate data repository containing a series of annual reports. If we want to see the annual report for the year 2000 we are unlikely to want to retrieve every report for every year and then manually search through those. We want to be able to specify the report we need. Therefore, if a highly simplified version of the data repository looked like Listing 5.1, we could use the position ( ) function to display the year 2000 report by using the <xsl:value-of> element with the following location path in the select attribute:

```
<xsl:value-of select="/AnnualReports/AnnualReport[position()=6]"/>
```

The location path works by selecting the AnnualReport element node, which has a context position of 6, and which has an AnnualReports element parent node (which itself is a child node of the root node).

Node Set Functions

The XPath node set functions allow us to use or manipulate node sets within an XPath expression.

count ( ) Function

The count ( ) function takes a single argument, a node set, and returns the number of nodes contained in the node set.

The count ( ) function can be used within XPath locations paths. For example, if we wanted to count the number of parent nodes in the context node, we could use the location path

```
count(..)
```

This means count the number of nodes that are the parent of the context node. Of course, since all nodes have only one parent node (except the root node that has none),

```
<?xml version='1.0'?>
<TOC>
<Chapters>
<Chapter number="1">Title to be decided.</Chapter>
<Chapter number="2">Title to be decided.</Chapter>
<Chapter number="3">Title to be decided.</Chapter>
<Chapter number="4">Title to be decided.</Chapter>
</Chapters>
<Appendices>
<!-- Will I include any appendices? -->
</Appendices>
</TOC>
```

Listing 5.2 An Outline for a Book Expressed in XML (TOC.xml).

the count ( ) function will return either one (most commonly) or zero (when the context node is the root node) nodes.

A node set is a mathematical set of distinct nodes. It doesn't matter if they yield the same string-value; they remain distinct nodes within the node set. For example, if while drafting the table of contents for this book I had decided to store an early draft in XML, I might have a document like that shown in Listing 5.2.

You can see that the draft had identical strings for the content of each <Chapter> element. However, if I used the count ( ) function to find out how many <Chapter> elements existed, with an expression like the following, the value returned would be four, despite the fact that the string-values of each of the four nodes are identical.

```
count(/TOC/Chapters/Chapter)
```

Let's create a simple example that uses the count ( ) function to add up the number of databases listed in a catalog of relational databases. Listing 5.3 shows the catalog of databases, shortened for convenience of presentation.

```
<?xml version='1.0'?>
<CatalogDatabases>
<Database>
<Name>DB2</Name>
<Manufacturer>International Business Machines</Manufacturer>
<Version>7</Version>
</Database>
<Database>
<Name>SQL Server</Name>
<Manufacturer>Microsoft Corporation</Manufacturer>
```

Listing 5.3 A Brief Catalog of Database Management Systems (DatabaseCatalog.xml).
(*continues*)

```
<Version>SQL Server 2000</Version>
</Database>
<Database>
<Name>Oracle</Name>
<Manufacturer>Oracle Corporation</Manufacturer>
<Version>9i</Version>
</Database>
</CatalogDatabases>
```

Listing 5.3 (*Continued*)

We want an XPath expression that will return the number of <Database> elements in the catalog and the count ( ) function provides what we need, as in the XSLT stylesheet shown in Listing 5.4.

Applying the stylesheet in Listing 5.4 to our XML source document yields an HTML document as shown in Figure 5.1.

One of the facilities missing from the XPath 1.0 specification is a way to test whether a particular node is present in two node sets. The count ( ) function provides a workaround for that omission. If you want to find out if a node set contains, for example, the parent of the context node, use the count ( ) function with the union of two node sets, with one node set being held in an XSLT variable:

```
count($nodeset | ..)
```

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


<xsl:template match="/">
<html>
<head>
<title>Number of databases in the catalog.</title>
</head>
<body>
The database catalog contains information on <xsl:value-of select="count
  (CatalogDatabases/Database)"/> relational
database management systems.
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 5.4 A Stylesheet to Demonstrate the count ( ) Function (DatabaseCatalog.xsl).

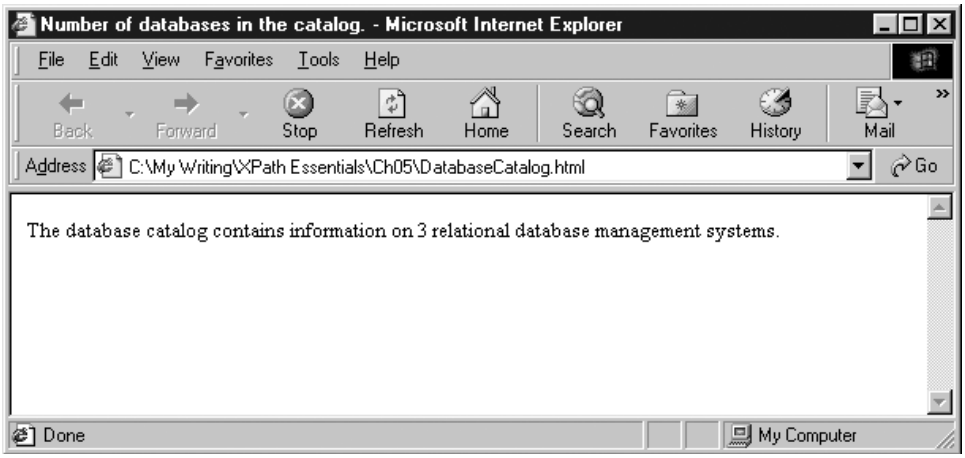

Number of databases in the catalog. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch05\DatabaseCatalog.html
Go
The database catalog contains information on 3 relational database management systems.
Done
My Computer

Figure 5.1 Using the count ( ) Function.

Then compare it with the node set itself as in the following expression:

```
<xsl:if test="count($nodeset | ..) = count($nodeset)>
<!-- the conditional action goes here. -->
</xsl:if>
```

If the result of the first count ( ) function equals the result of the second count ( ) function, then the parent node is already present in the variable $nodeset.

The count ( ) function can also be used to count how many unique values are present in a node set. If we wanted to know how many unique manufacturers of graphics software were present in the XML document in Listing 5.5, we could use the count ( ) function, but we would need to make sure that duplicates were identified and not included in the count. Remember that XPath has a preceding axis and that is very useful here.

Therefore, we can output the number of unique software manufacturers with the XSLT stylesheet in Listing 5.6.

```
<?xml version='1.0'?>
<GraphicsSoftware>
<Product>
<Manufacturer>Jasc</Manufacturer>
<ProductName>WebDraw</ProductName>
<Version>1.0</Version>
</Product>
<Product>
<Manufacturer>Adobe</Manufacturer>
<ProductName>Illustrator</ProductName>
```

Listing 5.5 A Brief Catalog of Graphics Software (GraphicsSoftware.xml).

(*continues*)

```
<Version>9</Version>
</Product>
<Product>
<Manufacturer>Macromedia</Manufacturer>
<ProductName>Fireworks</ProductName>
<Version>3.0</Version>
</Product>
<Product>
<Manufacturer>Jasc</Manufacturer>
<ProductName>Paint Shop Pro</ProductName>
<Version>7.0</Version>
</Product>
<Product>
<Manufacturer>Adobe</Manufacturer>
<ProductName>Photoshop</ProductName>
<Version>6.0</Version>
</Product>
</GraphicsSoftware>
```

Listing 5.5 (*Continued*)

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the count() function to count unique values</title>
</head>
<body>
<br />
<p>In the XML source file there are
<xsl:value-of
  select="count(/GraphicsSoftware/Product/Manufacturer[not(.=preceding::
  Product/Manufacturer)])"/>
  different manufacturers of graphics software.
</p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 5.6 A Stylesheet Using the count ( ) Function to Detect Uniqueness (GraphicsSoftware.xsl).

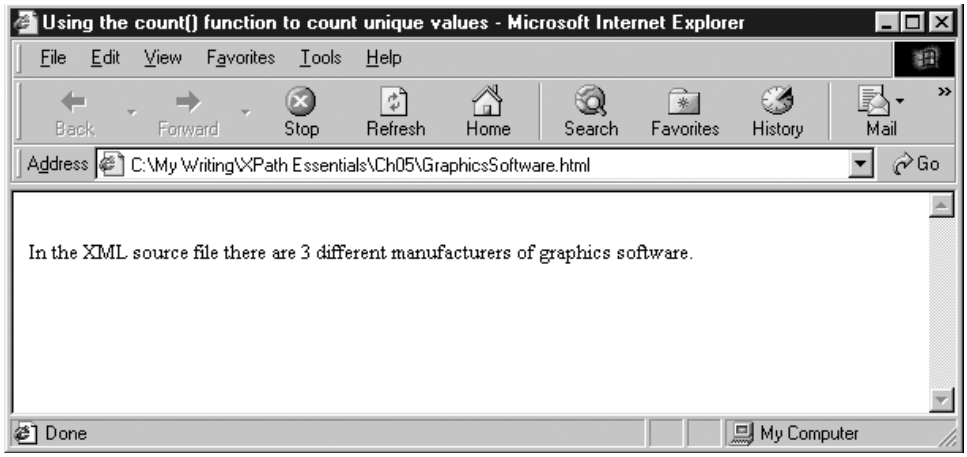

Using the count() function to count unique values - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch05\GraphicsSoftware.html
Go
In the XML source file there are 3 different manufacturers of graphics software.
Done
My Computer

Figure 5.2 Using the count ( ) Function to Identify Unique Nodes.

The output from the transformation is shown in Figure 5.2.

Another possible use for the count ( ) function is to determine the depth to which an element is nested. Let's suppose we had the source document shown in Listing 5.7.

We could determine the level of nesting of, for example, the <Level4Node> element using the stylesheet in Listing 5.8.

```
<?xml version='1.0'?>
<ElementNode>
 <Level2Node>
  <Level3Node>
   <Level4Node>
   </Level4Node>
  </Level3Node>
 </Level2Node>
</ElementNode>
```

Listing 5.7 A Simple Deeply Nested XML Document (ElementNode.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
```

Listing 5.8 Using the count ( ) Function to Determine Level of Nesting (ElementNode.xsl). (*continues*)

```
<html>
<head>
<title>Using the count() function to return the depth of a node</title>
</head>
<body>
<xsl:apply-templates
select="ElementNode/Level2Node/Level3Node/Level4Node"/>
</body>
</html>
</xsl:template>

<xsl:template match="Level4Node">
<p>The &lt;Level4Node&gt; is at a depth of <xsl:value-of
select="count(ancestor::*)"/>
 levels below the &lt;ElementNode&gt; document element.</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 5.8 (*Continued*)

TEAMFLY

The way this works is to count the number of ancestor nodes of the context node. The <xsl:apply-templates> element selects a <Level4Node> element to be the context node when the matching <xsl:template> element is instantiated.

id ( ) Function

The id ( ) function takes a single argument, which may be of any data type. A node set containing the required ID value(s) is returned. If the argument is a node set as in the following code, then the first node with the specified string as the value of an ID attribute that satisfies the remainder of the expression is the node that is returned.

```
id("AnID")
```

If a single string is sought for the ID attribute, then only a single node is returned. It is an error if more than one element in an XML document has the same value for an ID attribute.

NOTE **In XPath 1.0, an ID attribute for this purpose is any attribute that is declared in the Document Type Definition as having type ID.**

If the argument is of a data type other than a node set, then it is either a string or is converted to a string, according to the functionality of the string ( ) function. The string passed to the id ( ) function is treated as a whitespace-separated list of tokens representing ID values. Each value is tested, and for each matching node found for such a token, a node is added to the node set to be returned. Thus if the series of tokens

matches more than one node, then the number of nodes in the returned node set may be greater than one.

The id ( ) function returns a node set, often a single node, where an ID attribute matches the argument passed to the id ( ) function. It is not possible in a valid XML document for more than one element to have what ought to be a unique ID attribute value.

For example, if each stocked item had its own unique SKU (stock keeping unit) attribute as an ID attribute then the expression below would return the single matching <item> element where the SKU attribute had the value of "ABC-123".

```
id("ABC-123")
```

An attribute must be specified as being of an ID type in the DTD for the XML source document.

NOTE **The id ( ) function only works predictably when it is known that the source document(s) being processed by the XSLT/XPath processor have been validated against a DTD or schema. Since there is no requirement on an XSLT processor to process only valid documents, it may process nonvalid documents that have elements with the same ID attribute. In that case, the XPath specification indicates that only the first such node will be returned.**

The functionality of the id ( ) function can also be achieved by the location path

```
//*[@id="ABC123"]
```

But it is likely that the implementation of the id ( ) function will be more efficient than the location path with a predicate. Additionally, the key ( ) function could carry out the task of the id ( ) function but with the limitation that only one ID attribute may be searched for at a time.

The id ( ) function always locates elements in the same document as the context node. Thus if you want to search for an ID attribute in an external document, you need to change the context node first. For example, use the following document ( ) function:

```
<xsl:for-each select="document('external.xml')">
  <xsl:copy-of select="id('ABC123')"/>
</xsl:for-each>
```

Notice that the select attribute of the <xsl:for-each> element makes the root node of the external document, external.xml, the context node. Thereafter the id ( ) function searches within that same document for a node to match the desired argument passed to the id ( ) function.

last ( ) Function

The last ( ) function takes no arguments. It returns a number that corresponds to the context size.

Remember that in XPath the context has a context node, a context position, and a context size as well as variable mappings and a function library. In XSLT, the term *current node list* is used instead.

If our source document looked like Listing 5.9, we could select the last of the <Chapter> elements by using the location path

```
/Chapters/Chapter[position()=last()]
```

This would result in the node that represents the <Chapter> element whose text node has the value of "Fourth" being selected. In other words, the node representing the last <Chapter> element is selected.

The previous location path could also have been written as follows using the abbreviated syntax.

```
/Chapters/Chapter[last()]
```

But be careful not to fall into the potential trap of expecting that location path to return a Boolean value. The last ( ) function returns a number that corresponds to the context size.

local-name ( ) Function

The local-name function may take zero or one arguments. If no argument is passed to the local-name ( ) function, then the node set selected has one member—the context node. Otherwise, the argument represents a node set. The local-name ( ) function returns a string value representing the local part of the qualified name of a node—either the local part of the expanded name of the context node if no argument is passed, or the local part of the expanded name of the first node in document order of the node set specified in the argument passed to the function.

When the node is an element node or an attribute node, then the element type name (after any prefix and colon) is returned. If the root node, a comment node, or a text node were the context node, then the empty string would be returned. If the node was a processing instruction node, then the target is returned. If the node was a namespace node, the namespace prefix would be returned unless this represented the default namespace. In this case, the empty string would be returned.

```
<Chapters>
<Chapter>First</Chapter>
<Chapter>Second</Chapter>
<Chapter>Third</Chapter>
<Chapter>Fourth</Chapter>
</Chapters>
```

Listing 5.9 A Short List of Chapters (Chapters.xml).

For example, given the following element, the local part of the QName is "document", the namespace prefix is "xmml", and the namespace prefix is separated by a colon from the local part of the QName.

```
<xmml:document xmlns:xmml="http://www.xmml.com/Schemas/">
```

In Chapter 2, I explained how the local-name ( ) function returns the local part of the expanded name of the first node, and the first node only, in a node set. Thus if you wanted to find out, for example, the local part of the names of a series of elements, you would need to modify the approach.

Let's assume our source document was Listing 5.10 (I have used an all-uppercase namespace prefix here).

We can apply the stylesheet in Listing 5.11 to it.

```
<?xml version='1.0'?>
<XMML:Book xmlns:XMML="http://www.XMML.com/BookSchemas/">
<XMML:Introduction>Some introductory material</XMML:Introduction>
<XMML:Chapter>This is the first chapter</XMML:Chapter>
<XMML:Chapter>This is a second chapter</XMML:Chapter>
<XMML:Chapter>This is a third chapter</XMML:Chapter>
<XMML:Appendix>Some material of an appendix-like nature.</XMML:
  Appendix>
</XMML:Book>
```

Listing 5.10 A Simple Book Structure Using Namespaces. (XMMLBook.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:XMML="http://www.XMML.com/BookSchemas/">

<xsl:template match="/">
<html>
<head>
<title>Using the local-name()function</title>
</head>
<body>
<xsl:apply-templates select="XMML:Book/XMML:*"/>
</body>
</html>
```

Listing 5.11 A Stylesheet to Demonstrate the local-name ( ) Function (XMMLBook.xsl).
(*continues*)

```
</xsl:template>

<xsl:template match="XMML:Book | XMML:Introduction | XMML:Chapter |
  XMML:Appendix">
<p>The local part of the &lt;XMML:<xsl:value-of select="local-
  name(.)"/>&gt; element is <xsl:value-of
select="local-name(.)"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 5.11 (*Continued*)

The <xsl:apply-templates> element in the main template selects as individual node sets, each of the child element nodes of the element node that represents the <XMML:Book> element. The other <xsl:template> element in the stylesheet matches any of those elements:

```
<xsl:template match="XMML:Book | XMML:Introduction | XMML:Chapter |
  XMML:Appendix">
```

Thus, for each node set (each of one child element node) selected by the <xsl:apply-templates> element, the local name of the element is placed into the output text in two positions.

name ( ) Function

The name ( ) function takes zero or one arguments. If no argument is passed to the name ( ) function, then the node to which the function is applied is the context node. Otherwise, the argument passed to the function defines a node set to which the function is applied. The name ( ) function returns a string that is a QName representing the name of a node.

Often the string returned by the name ( ) function will be the name of the node as written in the XML source document. Thus, for the following element, if the name ( ) function were applied to the element node that represents it, then the string returned would be "xmml:document".

```
<xmml:document xmlns:xmml="http://www.xmml.com/Schemas/">
```

If the node is other than an element or attribute node, the value returned is as was described for the local-name ( ) function.

The name ( ) function can be used in any situation where you want to directly display the element type name in the output document. One situation where you might want to do this is when producing an error message. For example, if you were creating a book catalog such as that in Listing 5.12 and wanted to be sure that all books had a price attribute, you could use the following XSLT snippet (from BookCatalog.xsl not shown in full):

```
<xsl:if test="not(@Price)">
<xsl:message>A <xsl:value-of select="name()"/> element is lacking a
Price attribute.
</xsl:message>
<xsl:if>
```

Another possible use of the name ( ) function is when you are confronted with an XML document for which either no schema exists or no schema is available and you want to find out what elements are used in the document. For short documents this can easily be done by eye, but for lengthy documents an XSLT stylesheet can be used to generate a list of all the element names used in the mystery document.

namespace-uri ( ) Function

The namespace-uri ( ) function takes zero or one arguments. It returns a string whose value is the namespace URI of the namespace in the expanded name of the node in question. If no argument is passed to the namespace-uri ( ) function, then the node evaluated is the context node. If an argument is passed to the function and it is a node set, then the first node in document order contained in the node set is evaluated.

The string returned depends on the type of node being evaluated. If it is the root node, a text node, a processing instruction node, a comment node, or a namespace node, an empty string is returned. If the node is an element node or an attribute node, then the namespace URI is returned by the namespace-uri ( ) function as a string.

If we again used the following source document that we saw earlier in Listing 5.10, we could apply the XSLT stylesheet in Listing 5.13 to output the namespace URI for each element node that is a child of the <XMML:Book> element.

```
<?xml version='1.0'?>
<XMML:Book xmlns:XMML="http://www.XMML.com/BookSchemas/">
<XMML:Introduction>Some introductory material</XMML:Introduction>
<XMML:Chapter>This is the first chapter</XMML:Chapter>
<XMML:Chapter>This is a second chapter</XMML:Chapter>
<XMML:Chapter>This is a third chapter</XMML:Chapter>
<XMML:Appendix>Some material of an appendix-like nature.</XMML:Appendix>
</XMML:Book>
```

The <xsl:apply-templates> element in the main template matches all the child elements. For each of those, the namespace URI is output, as you can see in Figure 5.3.

```
<?xml version='1.0'?>
<BookCatalog>
<Book Price="44.99">XPath Essentials</Book>
<Book >XML Schema Essentials</Book>
</BookCatalog>
```

Listing 5.12 A Skeleton Book Catalog in XML (BookCatalog.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:XMML="http://www.XMML.com/BookSchemas/">

<xsl:template match="/">
<html>
<head>
<title>Using the namespace-uri()</title>
</head>
<body>
<xsl:apply-templates select="XMML:Book/XMML:*"/>
</body>
</html>
</xsl:template>

<xsl:template match="XMML:Introduction | XMML:Chapter | XMML:Appendix">
<p>The namespace URI of the &lt;XMML:<xsl:value-of select="local-
  name(.)"/>&gt; element is <xsl:value-of
select="namespace-uri(.)"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 5.13 A Stylesheet to Demonstrate the namespace-uri ( ) Function (XMMLBook2.xsl).

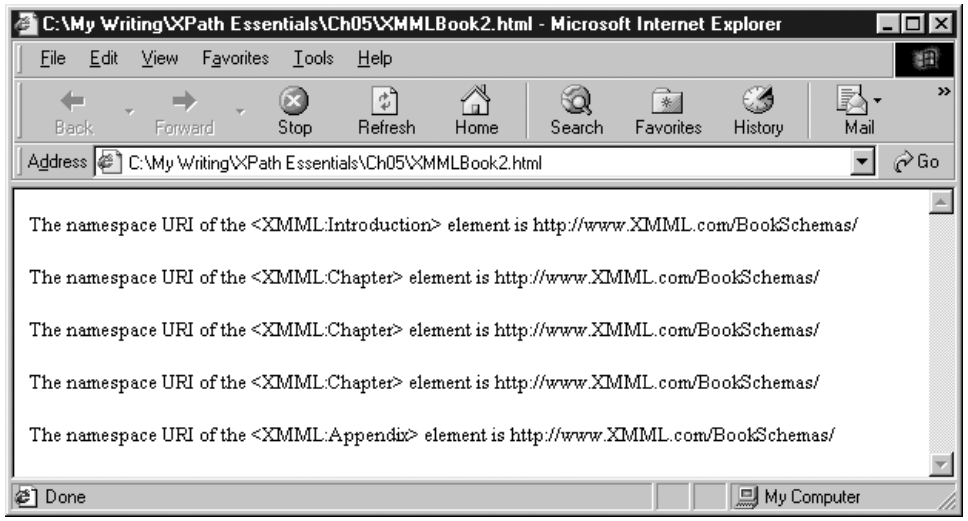


Figure 5.3 Demonstration of the namespace-uri ( ) Function.

position ( ) Function

The position ( ) function selects a node or node set according to criteria relating to the context position(s) of the nodes. If we apply the XSLT stylesheet in Listing 5.14 to Listing 5.9, we can output the content of the selected node to confirm that the <XMML:Chapter> in position 3 has been selected.

The output is shown in Figure 5.4.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns:XMML="http://www.XMML.com/BookSchemas/">

<xsl:template match="/">
<html>
<head>
<title>Using the position()function</title>
</head>
<body>
<xsl:apply-templates select="XMML:Book/XMML:Chapter[position()=3]"/>
</body>
</html>
</xsl:template>

<xsl:template match="XMML:Chapter">
<p>The content of the selected &lt;XMML:<xsl:value-of select="local-
  name(.)"/>&gt;
element is <xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 5.14 A Stylesheet to Demonstrate the position ( ) Function (XMMLBook4.xsl).

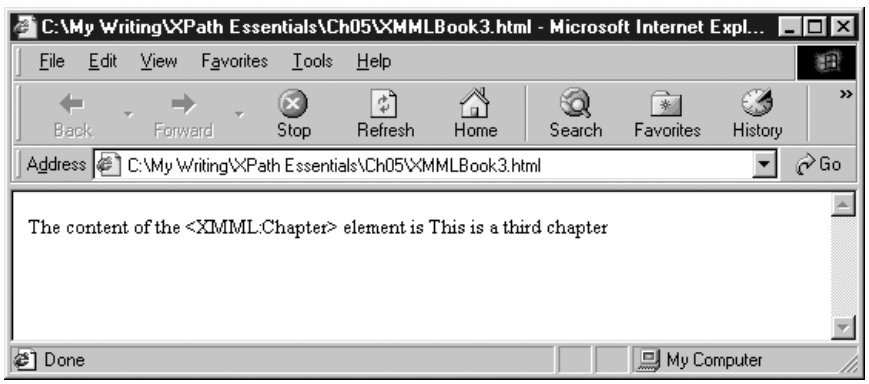

C:\My Writing\XPath Essentials\Ch05\XMMLBook3.html - Microsoft Internet Expl...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address C:\My Writing\XPath Essentials\Ch05\XMMLBook3.html
Go
The content of the <XMML:Chapter> element is This is a third chapter
Done
My Computer

Figure 5.4 Using the position ( ) Function.

Number Functions

XPath 1.0 provides a limited number of number functions that allow us to perform fairly simple mathematical calculations.

ceiling ( ) Function

The ceiling ( ) function takes a single argument, which is a number, and returns the smallest integer, which is greater than or equal to the numeric value of the argument.

If the argument passed to the ceiling ( ) function is not a number, then the value returned will be NaN, meaning "not a number." If the argument is positive infinity or negative infinity then the value returned will be the same as the argument.

If the argument passed to the ceiling ( ) function is an integer, then the value returned will be the same integer. Thus

```
ceiling(5)
```

returns 5. However if the argument is a real number,

```
ceiling(1.7)
```

then the next highest integer (in this case, 2) is returned.

If the argument passed to the ceiling ( ) function is a node set, then the function is applied to the value of the first node, in document order, of the node set.

One possible use for the ceiling ( ) function is in the construction of an HTML table to display a set of values in multiple columns. For example, if the number of values to be displayed was contained in a variable $vtbd and you wanted to display those values in four columns, then the number of rows required could be determined in this way:

```
ceiling(count($vtbd) div 4)
```

Let's suppose the value of the variable $vtbd is 7. When divided by 4, the result is 1.75. Applying the ceiling ( ) function to 1.75 results in a value of 2 being returned from the ceiling ( ) function.

floor ( ) Function

The floor ( ) function takes a single argument that is of the number type. The floor ( ) function returns the largest integer, which is less than or equal to the value of the argument passed to the function. If the argument is not a number, then it is first converted to a number using the number ( ) function.

Effectively, the floor ( ) function rounds a number down to the next lower integer, if it is not an integer. In the latter case, the value returned is the value passed as an argument to the function.

For example,

```
floor(3.3)
```

would return an integer value of 3.

```
floor(20)
```

would return an integer value of 20.

If the argument passed to the function is a node set, then the value of the first node, in document order, in the node set will determine the value of the argument to which the function is applied. If the argument passed to the function is a string that cannot be converted to a number, then the function returns NaN.

number ( ) Function

The number ( ) function takes zero or one arguments. If no argument is passed, then the number ( ) function is applied to the context node. If an argument is passed, then it is converted to a number.

round ( ) Function

The round ( ) function rounds a floating point number to the closest integer.

sum ( ) Function

The sum ( ) function determines the sum of the nodes in the node set that is its argument.

String Functions

XPath 1.0 provides a few functions that allow us to manipulate strings.

concat ( ) Function

The concat ( ) function takes two or more string arguments. The concat ( ) function concatenates the two (or more) strings and returns the concatenated string.

If one (or more) of the arguments is not a string, then the string ( ) function (described later in the chapter) is applied to the argument to convert it into a string. The concat ( ) function is then applied, as before, to the two (or more) string arguments.

If we wanted to concatenate the two strings “Hello” and “World!” using the concat ( ) function then we could do so like this

```
concat("Hello ", "World!")
```

and the string “Hello World!” would be returned by the concat ( ) function.

The concat ( ) function provides a useful alternative to using multiple <xsl:value-of> elements. For example, if we wanted to display the customer name and date of a particular invoice in this format “Invoice To: [Customer name goes here] [Date goes here], then we could do so using the following, which concatenates four strings:

```
<xsl:value-of select="concat("Invoice To: ", CustomerName, " ",
  DateOfInvoice)/>
```

The first "Invoice To:" and the third " " are passed to the function as strings. The second and fourth arguments are passed to the concat ( ) function as node sets (of one node), the results being automatically converted to a string and then concatenated.

If we wanted to achieve the same output without using the concat ( ) function, we would have needed code like this:

```
<xsl:text>Invoice To: </xsl:text>
<xsl:value-of select="CustomerName"/>
<xsl:text> </xsl:text>
<xsl:value-of select="DateOfInvoice"/>
```

Another situation where the concat ( ) function can be useful is in relation to defining XSLT keys.

The concat ( ) function can also be used to produce output such as a comma separated list. For example, if we had a source XML document like that in Listing 5.15, which we wanted to display as a comma separated list, we could use the concat ( ) function within an XSLT stylesheet like that in Listing 5.16.

TRAMFLY

```
<?xml version='1.0'?>
<Cameras>
<Camera brand="Canon" model="T90"/>
<Camera brand="Praktica" model="LLC"/>
<Camera brand="Nikon" model="F"/>
<Camera brand="Pentax" model="M7"/>
<Camera brand="Pentacon" model="Six TL"/>
</Cameras>
```

Listing 5.15 A Short Catalog of Camera Brands (Cameras.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output indent="yes"/>
<xsl:template match="/">
  <CameraCollection>
  <xsl:for-each select="//Camera">
  <Camera><xsl:value-of select="concat(@brand, ', ', @model)"/></Camera>
  </xsl:for-each>
  </CameraCollection>
</xsl:template>

</xsl:stylesheet>
```

Listing 5.16 A Stylesheet to Demonstrate the concat ( ) Function (Cameras.xsl).

```
<?xml version="1.0" encoding="utf">
<CameraCollection>
  <Camera>Canon, T90</Camera>
  <Camera>Praktica, LLC</Camera>
  <Camera>Nikon, F</Camera>
  <Camera>Pentax, M7</Camera>
  <Camera>Pentacon, Six TL</Camera>
</CameraCollection>
```

Listing 5.17 The Output from the Stylesheet in Listing 5.16 (CamerasOut.xml).

This will produce output like that shown in Listing 5.17.

contains ( ) Function

The contains ( ) function takes two string arguments and tests whether the first string argument contains the second string argument. If the second string argument is contained in the first string argument, then "true" is returned; otherwise, "false" is returned. If one or both of the arguments is not passed as a string, then the argument(s) undergo an automatic conversion to string before evaluation.

For example, to test whether the string "And" existed within the string "Andrew" we could use the contains ( ) function like this

```
contains("Andrew", "And")
```

In this case, the contains ( ) function would return true.

In the odd situation where the second argument is the empty string (or an argument of another type that evaluates to an empty string), the contains ( ) function always returns "true". If the first argument is the empty string (or is converted to the empty string, in the case of the first argument not being a string) then the contains ( ) function returns "false" except in the fairly bizarre situation that the second argument is also the empty string (or converts to that), in which case "true" is returned.

For example, if we wanted to test whether any of the books in the XML Essentials series represented by the XML document in Listing 5.18 had an <Author> element that contained the string "Watt", we could use the contains ( ) function in an XSLT stylesheet as in Listing 5.19.

This stylesheet would generate an HTML results document like that shown in Figure 5.5.

```
<?xml version='1.0'?>
<BookCatalog>
<Book Price="44.99">
<Title>XSL Essentials</Title>
```

Listing 5.18 A Brief Book Catalog Expressed in XML (BookCatalogAuthor.xml).

(*continues*)

```
<Authors>
  <Author>Michael Fitzgerald</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XHTML Essentials</Title>
<Authors>
  <Author>Michael Sauers</Author>
  <Author>R. Allen Wyke</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XPath Essentials</Title>
<Authors>
  <Author>Andrew Watt</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XML Schema Essentials</Title>
<Authors>
  <Author>Andrew Watt</Author>
  <Author>R. Allen Wyke</Author>
</Authors>
</Book>
</BookCatalog>
```

Listing 5.18 (*Continued*)

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>XML Esssentials Series with author "Watt"</title>
</head>
<body>
<h1>The Wiley XML Essentials Series</h1>
<h3>The following books in the series were written by an author whose
  name was Watt.</h3>
 <xsl:apply-templates/>
</body>
</html>
```

Listing 5.19 A Stylesheet to Demonstrate the contains ( ) Function (BookCatalogAuthor.xsl). (continues)

```
</xsl:template>

<xsl:template match="Book">
  <xsl:if test="contains(Authors/Author, 'Watt')">
  <br />
  <xsl:value-of select="Title"/><br />
  <xsl:for-each select="Authors/Author">
  <xsl:value-of select="."/><br />
  </xsl:for-each>
  <xsl:text>$</xsl:text><xsl:value-of select="@Price"/><br />
  </xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Listing 5.19 (*Continued*)

normalize-space ( ) Function

The normalize-space ( ) function takes zero or one arguments. If no argument is passed to the normalize-space ( ) function, then the string-value of the context node is evaluated. Otherwise the argument is a string or is converted to a string and then evaluated. The purpose of the normalize-space ( ) function is to remove leading and trailing whitespace

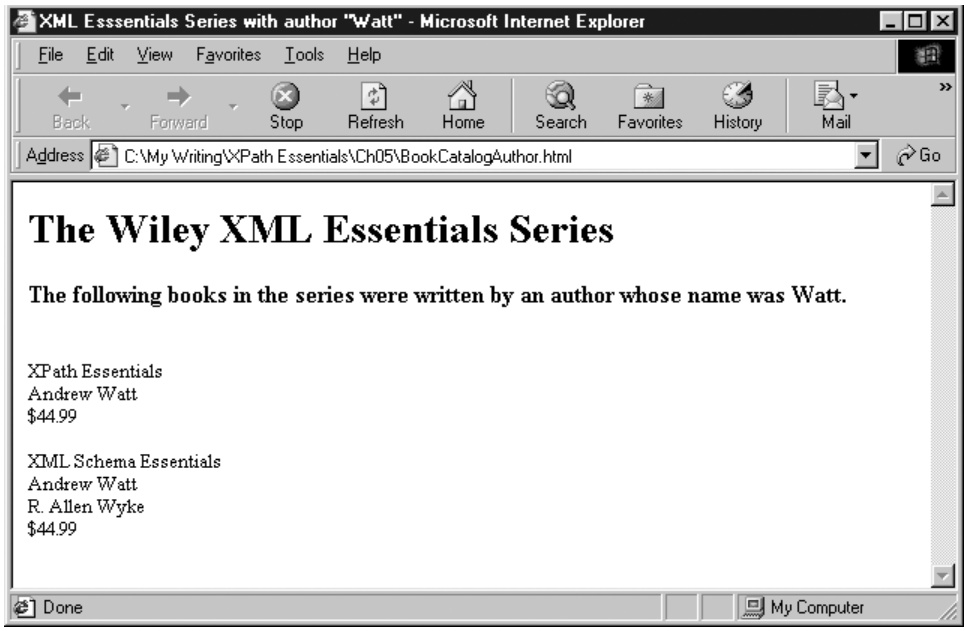

XML Esssentials Series with author "Watt" - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch05\BookCatalogAuthor.html
Go
The Wiley XML Essentials Series
The following books in the series were written by an author whose name was Watt.
XPath Essentials
Andrew Watt
$44.99
XML Schema Essentials
Andrew Watt
R. Allen Wyke
$44.99
Done
My Computer

Figure 5.5 Using the contains ( ) Function.

from a string and to standardize the whitespace within a string by replacing any internal sequences of whitespace characters by a single space character. It returns a string with the features just described.

The XML specification defines whitespace as a sequence of space, tab, newline, and carriage return characters. Often when processing a source document, it is useful to remove leading and trailing whitespace and ensure that multiple whitespace characters are removed. The normalize-space ( ) function achieves that.

NOTE The <xsl:strip-space> element does not have the same effect as the normalize-space ( ) function. The <xsl:strip-space> element removes text nodes where the content is exclusively whitespace. It has no effect, for example, on text nodes that contain leading or trailing whitespace or sequences of whitespace characters if characters that are not whitespace are also contained in the text node.

The normalize-space ( ) function can be useful at times, but you must be careful how you apply it. For example, it could have undesired effects when processing an XHTML source document, for example. Typically in XHTML, which includes <i> or <b> elements to control presentation, the normalize-space ( ) function may produce leading or trailing whitespace in text as follows:

```
<p>Leading and trailing white space here is <i>very</i> important in
  determining the desired layout.</p>
```

If the normalize-space ( ) function was applied to XHTML such as that just shown, the leading and trailing whitespace would be stripped out with an undesirable juxtaposition of words.

starts-with ( ) Function

The starts-with ( ) function takes two string arguments and tests whether the first string argument starts with the second. It returns a Boolean value.

string ( ) Function

The string ( ) function can convert an argument, whether it be of type node set, number, or Boolean to a string. The effect of the string ( ) function depends on the type of its argument.

string-length ( ) Function

The string-length ( ) function takes a single string argument and returns the number of characters contained in the string. Thus the following code would return 4.

```
string-length("Mary")
```

substring ( ) Function

The substring ( ) function takes three arguments, the first of which is a string, and returns a substring determined by the second and third arguments. Thus the following code would return the substring "and". The second argument indicates the first character to be included in the substring and the third argument indicates the length of the substring.

```
substring("colander", 4, 3)
```

substring-after ( ) Function

The substring-after ( ) function takes two string arguments and returns the part of the first argument that is left after the first occurrence within the first argument of the second string argument.

substring-before ( ) Function

The substring-before ( ) function takes two string arguments and returns the part of the first argument that occurs before the first occurrence within the first argument of the second string argument.

translate ( ) Function

The translate ( ) function can be used to achieve case conversion.

Boolean Functions

XPath provides several functions to allow the manipulation of Boolean values.

boolean ( ) Function

The boolean ( ) function takes a single argument and converts it to a Boolean value. The argument may be a number, string, Boolean, or node set.

If the argument is a number, then the number is converted to a Boolean.

```
boolean(1)
```

If the argument has the value of zero, or NaN, the Boolean value is false. For any other number the Boolean value is true.

If the argument is a string, then the returned value is true if the string is anything other than an empty string.

```
boolean("Hello")
```

If the string argument is the empty string, the resulting Boolean value is false.

If the argument of the boolean ( ) function is itself a Boolean value, then that value is unchanged.

If the argument of the boolean ( ) function is a node set, then the returned value is false if the node set is empty.

```
boolean(//Chapter)
```

In the example above, if there is one or more <Chapter> elements in the document, then the boolean ( ) function returns true, since the node set is not empty.

The boolean ( ) function could be used like this in XSLT:

```
<xsl:if test="boolean(//Chapter)">
<xsl:message>This document contains one or more Chapter elements.
</xsl:message>
</xsl:if>
```

The test attribute of the <xsl:if> element has a value determined by calling the boolean ( ) function to evaluate whether the document contains one or more nodes representing a <Chapter> element. If it does, the value of the test attribute is "true" and therefore the content of the <xsl:message> element is expressed.

false ( ) Function

The false ( ) function does not take any arguments. XPath does not provide any Boolean constants. The false ( ) and true ( ) functions, respectively, return the Boolean values of "false" and "true" and so can be used in situations where a Boolean constant might otherwise be required.

The uses of the false ( ) function are relatively few. One possible use is when you want to deactivate a section of an XSLT stylesheet while debugging. It is possible to use XSLT comments to do that, but comments suffer from the limitation that they can't be nested so it can be a tedious process to use them. The use of the false ( ) function provides an alternative approach.

```
<xsl:if test="false()">
 <!-- Nothing in here gets processed. -->
</xsl:if>
```

The content nested within an <xsl:if> element is processed only when the value of the expression contained in the test attribute evaluates to "true". By using the false( ) function to define the value of the test attribute, we can be sure that it will evaluate to false and so any code nested within the <xsl:if> element is effectively commented out.

The <xsl:if> elements can be nested like this:

```
<xsl:if test="false()">
  <!-- Nothing in here gets processed. -->
    <xsl:if test="false()">
      <!-- This element provides a separate nested switch -->
    </xsl:if>
</xsl:if>
```

If you tried to do something similar using multiple comments, the closing part of the nested comment would have ended both comments. So when a processor encounters the remainder of the outer comment, an error is highly likely, especially when the processor encounters the "—>", which would have closed the outer comment.

lang ( ) Function

The lang ( ) function takes a single string argument, or if the argument is not a string it is converted to a string according to the behavior of the string ( ) function. The lang ( ) function returns a Boolean value. The purpose of the lang ( ) function is to test whether or not the language of the context node, as determined by the relevant xml:lang attribute, matches the language passed to the lang ( ) function as its argument.

If the context node itself possesses an xml:lang attribute, then that is the relevant value tested by the lang ( ) function. Otherwise, it is the value of the xml:lang attribute on the ancestor node closest to the context node (that possesses an xml:lang attribute) that is applicable.

For example, if the following

```
<Chapter xml:lang="en-US">
```

which indicates the USA variant of English, then if tested against the expression

```
lang("en")
```

the result returned by the lang ( ) function would be the Boolean value of "true".

Thus in the following example document

```
<Chapter xml:lang="en" >I went to Germany and wanted to practice my
  German. So I walked up to a friendly looking man and said
<Quote xml:lang="de">Entschuldigen Sie bitte. Ich bin auslander.
<Translation xml:lang="en">Excuse me. I am a foreigner.</Translation>
</Quote>
and he said
<Quote xml:lang="fr">Pardon! Je ne comprends pas
<Translation xml:lang="en">Pardon me. I don't understand</Translation>
</Quote>
which was a little embarassing</Chapter>
```

the lang ( ) function would return true for

```
lang('en')
```

when the context node represented the <Chapter> element except when within the <Chapter> element the xml:lang attribute on the nodes representing the <Quote> elements changed the value of the xml:lang attribute to "de" (German) or "fr" (French), with the exception of those parts of the <Quote> elements contained within the <Translation> elements.

The lang ( ) function allows language-dependent processing of source XML documents to be carried out, assuming that the language of the source document (or its

parts) is adequately and accurately decorated with the xml:lang attribute. For example, if months were stored as numbers in the source document, we could output the name of the month rather than the number by using the lang ( ) function to determine which of a choice of processing should be carried out.

not ( ) Function

The not ( ) function takes one argument. The not ( ) function negates the condition passed in the argument and so if the argument evaluated to "true", then the value returned by the not ( ) function evaluates to a Boolean value of "false". Similarly if the argument evaluates to "false" the not ( ) function will return "true".

Should the argument passed to the function not be a Boolean value, then it is converted to a Boolean value using the rules applicable to the boolean ( ) function.

true ( ) Function

The true ( ) function returns the Boolean value "true".

XSLT Functions

The XSLT Recommendation provides a number of functions in addition to those in the XPath core function library. Since the main current use of XPath is with XSLT, we will look briefly at how these XSLT additional functions are used.

XSLT current ( ) Function

The current ( ) function is an XSLT function that takes no argument and returns a node set consisting only of the current node. The term *current node* is an XSLT term, which is not identical to the XPath concept of *context node*.

The purpose of the current ( ) function is to allow you to determine, and use, the current node when it differs from the context node.

The definition of the XPath context node is simple. It is the node returned by the XPath location path

```
.
```

or in its unabbreviated form

```
self::node()
```

The current node is an altogether more complex animal. The following are characteristics of the XSLT current node:

- When a global variable is being evaluated, the current node is the root node of the source document.
- When <xsl:apply-templates> is used to process a set of nodes, each node successively becomes the current node. If <xsl:apply-templates> is used without a select

attribute, then the nodes processed are the children of the previously current node. As the set of nodes is processed, the current node changes to the node being processed. After the set of nodes has been processed, the current node reverts to the node that it was before <xsl:apply-templates> was used.

- When <xsl:for-each> is used, each of the nodes within the <xsl:for-each> element in turn becomes the current node. After processing of the <xsl:for-each> element has completed, then, as with the <xsl:apply-templates>, the current node reverts to the node that it was before the <xsl:for-each> was applied.

NOTE The current node changes as described above when using <xsl:apply-templates> but the <xsl:call-template> and <xsl:apply-imports> elements do not cause the current node to be changed.

As mentioned earlier, the purpose of the current ( ) function is to allow you to determine and use the current node where it differs from the context node. Thus, as indicated above, when within an <xsl:for-each> element, the current node changes, although the context node does not.

XSLT document ( ) Function

The XSLT document ( ) function optionally takes one or two arguments. It allows an XSLT stylesheet to access external documents other than the XML source document. The document ( ) function may be used to access a single external XML document or a set of such documents.

The document ( ) function accesses the external XML file by resolving a URI reference (which is its first argument), it parses the XML of the external document into a tree structure and, if the URI reference does not contain a fragment identifier, returns the root node of the external document.

The URI reference (the first argument of the document ( ) function) may optionally include a fragment identifier. In that case a node other than the root node is returned as determined by the fragment identifier used. At present there is variation between XSLT processors as to how a fragment identifier is handled. Once the XPointer specification (see Chapter 9, "Using XPath in XPointer") is finalized, then it is likely that XSLT processors will have a more standardized way of processing fragment identifiers.

When the document ( ) function has two arguments, the first is a URI reference and the second is a base URI. The base URI of the first node in the second argument passed to the function is the basis for resolving any relative URI in the first argument. If, for example, the document ( ) function was called using

```
document(Called.xml)
```

then the Called.xml file would be a relative URI, with the Called.xml file being located in the same directory as the XSLT stylesheet that contained the document ( ) function.

The document ( ) function is expected to locate an external document and return an appropriate node. If, however, the URI cannot be resolved or the resource to which the

URI refers is not an XML document, then the XSLT Recommendation allows the processor to optionally signal an error or return an empty node set.

Let's look at a brief example of how the document ( ) function works in practice. Suppose you were a photographer interested in a particular camera and wanted to justify its purchase to your company. You might create a report that included the text of a couple of authoritative reports on the camera. Your source document might contain information about the camera with references to the reviews you want to quote, such as that shown in Listing 5.20.

In your report you might want to incorporate verbatim the two reviews that you were interested in. You could achieve this by using the document ( ) function in a stylesheet like that shown in listing 5.21.

Notice the use of the document ( ) function in the <xsl:copy-of> element, which causes the content of the two dummy files, PPEOS1review.xml,

```
<?xml version='1.0'?>
<Article>
This is the placeholder text for the Practical Photography review.
</Article>
```

```
<?xml version='1.0'?>
<Camera>
<Manufacturer>Canon</Manufacturer>
<Model>EOS1</Model>
<Review date="March 2001" source="Practical Photography"
  href="PPEOS1review.xml"/>
<Review date="December 2000" source="Professional Photographer"
  href="PPerEOS1review.xml"/>
</Camera>
```

TEAMFLY

Listing 5.20 A Short Catalog of Camera Reviews (CameraReviews.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html"/>

<xsl:template match="/">
<html>
<head>
<title>Using the document() function</title>
</head>
<body>
```

Listing 5.21 A Stylesheet to Demonstrate the document ( ) Function (CameraReviews.xsl).

```
<p>The Canon EOS1 is a great camera.</p>
<p>Here are two helpful reviews from reviewers who know what they are
  talking about.</p>
<xsl:apply-templates select="Camera/Review"/>
</body>
</html>
</xsl:template>

<xsl:template match="Review">
<xsl:for-each select=".">
<p>The following review appeared in <xsl:value-of
  select="@source"/>:</p>
<p><xsl:copy-of select="document(@href)"/></p>
</xsl:for-each>
</xsl:template>

<xsl:template match="Camera">
</xsl:template>
</xsl:stylesheet>
```

Listing 5.21 (*Continued*)

and, PPerEOS1review.xml,

```
<?xml version='1.0'?>
<Article>
This is the placeholder text for the Professional Photographer review.
</Article>
```

to be accessed by the document ( ) function and the content of the <Article> element to be copied to the output HTML file. The screen appearance of the HTML output file is shown in Figure 5.6.

XSLT element-available ( ) Function

This XSLT function takes a single argument, which is either a string or is converted automatically to a string. The purpose of the element-available ( ) function is to determine whether a particular XSLT element or extension element is available for use. The function returns a Boolean value.

The string argument must evaluate to an XML QName (that is, an XML name with an optional namespace prefix). The namespace prefix must correspond to a namespace declaration that is in scope in the part of the XSLT stylesheet where the element-available ( ) function is called.

The element-available ( ) function returns true if and only if the expanded-name is the name of an instruction. If the namespace URI is equal to the XSLT namespace URI, then it refers to an element defined in XSLT. If the namespace URI is anything other than the

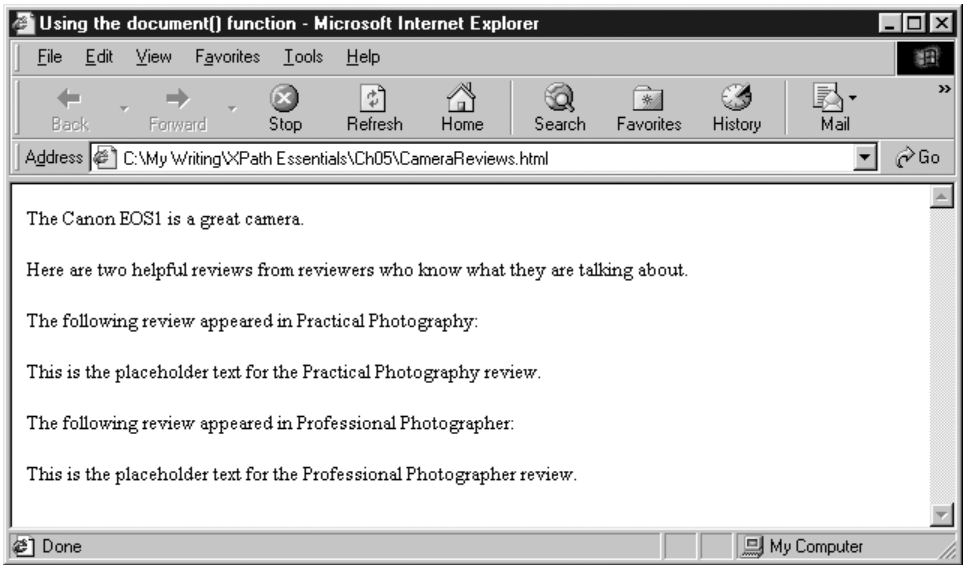

Using the document() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch05\CameraReviews.html
Go
The Canon EOS1 is a great camera.
Here are two helpful reviews from reviewers who know what they are talking about.
The following review appeared in Practical Photography:
This is the placeholder text for the Practical Photography review.
The following review appeared in Professional Photographer:
This is the placeholder text for the Professional Photographer review.
Done
My Computer

Figure 5.6 Using the document ( ) Function.

XSLT namespace URI, then it refers to an extension element. If that extension element is available, the function returns "true"; otherwise, it returns "false".

The usefulness of the element-available ( ) function is likely to increase as XSLT processors that handle versions other than or in addition to XSLT 1.0 become available. The Working Draft of XSLT 1.1, now discarded, proposed that a new <xsl:document> element be added. The element-available ( ) could have been used to test for elements like that, which might not be available for use to all XSLT processors.

XSLT format-number ( ) Function

The XSLT format-number ( ) function takes two or three arguments. The purpose of the format-number ( ) function is to convert numbers into strings for display. The function returns a string value.

When the format-number ( ) function takes two arguments, the first argument is a number and the second a string. The first argument is either a number, or if it is not a number it is first converted to a number using the XPath number ( ) function. The second argument is a string, which defines the format pattern. In the absence of a third argument, the default decimal format rules are applied.

A third argument may be passed to the function, in which case that argument must be a QName, an XML name with an optional namespace prefix, which is associated with a previously defined decimal format created using the <xsl:decimal-format> element.

The format pattern string is expressed in the syntax detailed in the Java Development Kit, JDK, 1.1. Detailed consideration of that syntax is beyond the scope of this chapter. Further information is found in Chapter 12 of the XSLT Recommendation (located at www.w3.org/TR/1999/REC-xslt-19991116).

NOTE **The format-number ( ) function has some functional overlap with the <xsl:number> element. However, the two are separate and make use of a different syntax.**

XSLT function-available ( ) Function

The function-available ( ) function takes one argument and returns a Boolean value. The argument is a string, which takes a value of a system (XPath or XSLT) function or an extension function of interest. The function is used to test whether a particular function is available to be used, whether that function is part of the standard library of XPath and XSLT functions, or whether that function is an XSLT extension function.

The syntax is like this

```
function-available(string)
```

If the string ( ) function is available, a Boolean value of "true" is returned by the function-available ( ) function. Note that in the argument passed to the function, the parentheses of the string ( ) function are omitted.

The string argument is a QName. If there is no prefix or the namespace URI is null, then the function-available ( ) function tests whether one of the standard XPath or XSLT functions is available. If the string argument is the name of any of the XPath or XSLT functions described in this chapter then the function-available ( ) function returns a value of "true", assuming that the XSLT processor is a conforming one.

Until such time as XSLT 2.0 appears, this function may be little used. But if you wanted to test at a future date for the hypothetical FancyNew ( ) function, then you could do something like this

```
<xsl:if test="function-available('FancyNew')">
<!-- Continue processing if the FancyNew() function is supported -->
</xsl:if>
```

NOTE **To use the <xsl:if> element with a possibly unavailable function, you will need to enable forwards-compatible processing by an XSLT processor by setting the version attribute in the <xsl:stylesheet> or <xsl:transform> element to a value greater than 1.0.**

At one time it was anticipated that this function would be used with XSLT 1.1, development of which has now been abandoned by the W3C. It is unlikely to be particularly useful until the XSLT 2.0 specification is finalized.

XSLT generate-id ( ) Function

The generate-id ( ) function takes one optional argument (that is, it may have zero or one arguments) and returns a string. The purpose of the generate-id ( ) function is to generate a string ID for a node set. After the generate-id ( ) function has run, then a node in that node set will have its own unique ID.

When the generate-id ( ) function is used without an argument, then the node set on which it operates is the context node.

When a node set is passed as the argument to the generate-id ( ) function, then the function operates on the first node in the node set, in document order.

NOTE **The IDs generated using the generate-id ( ) function bear no relation to any id attributes that may exist on elements within the source document. Thus the id ( ) function cannot be used to locate individual nodes.**

Let's use the generate-id ( ) function to create links between members of a list of HTML editors to be displayed in an HTML page and comments and/or assessments of them. The source code is shown in Listing 5.22.

```
<?xml version='1.0'?>
<HTMLEditors>
<HTMLEditor>
<Manufacturer>Allaire Corporation</Manufacturer>
<Product>HomeSite</Product>
<Version>4.5</Version>
<Comments>This is a placeholder for comments about Allaire HomeSite
  version 4.5.</Comments>
</HTMLEditor>
<HTMLEditor>
<Manufacturer>Macromedia</Manufacturer>
<Product>DreamWeaver</Product>
<Version>4</Version>
<Comments>This is a placeholder for comments about Macromedia
  Dreamweaver version 4.</Comments>
</HTMLEditor>
<HTMLEditor>
<Manufacturer>NetObjects</Manufacturer>
<Product>NetObjects Fusion</Product>
<Version>5</Version>
<Comments>This is a placeholder for comments about NetObjects Fusion
  version 5.</Comments>
</HTMLEditor>
<HTMLEditor>
<Manufacturer>Microsoft Corporation</Manufacturer>
<Product>FrontPage</Product>
<Version>2000</Version>
<Comments>This is a placeholder for comments about Microsoft FrontPage
  2000.</Comments>
</HTMLEditor>
</HTMLEditors>
```

Listing 5.22 A Short Catalog of HTML Editors (HTMLEditors.xml).

The XSLT stylesheet in Listing 5.23 can be used to generate an ID for the <Comments> elements in the source document. This allows us to link from a list at the early part of an HTML page where we simply give the names of the individual HTML editors to the further information on each contained in the <Comments> element in the source XML document.

The HTML output file, formatted for readability, looks like Listing 5.24.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
<html>
<head>
<title>Using the generate-id() function to link software titles to
  descriptions.</title>
</head>
<body>
<h2>HTML Editors</h2>
<xsl:for-each select="//HTMLEditor">
 <xsl:sort select="Manufacturer" order="ascending"
data-type="text"/>
 <h3><xsl:value-of select="Manufacturer"/>
 <xsl:text>: </xsl:text></h3>
 <p>
 <xsl:value-of select="Product"/>
 <xsl:text> - </xsl:text>
 <xsl:value-of select="Version"/></p>
 <p>For further information on <xsl:value-of select="Product"/>
  click here:
 <a href="#{generate-id(..Comments)}"><xsl:value-of
  select="Product"/></a>
 </p>
</xsl:for-each>
<xsl:for-each select="//HTMLEditor/Comments">
 <h3><a name="{generate-id()}">
 <xsl:value-of select="../Manufacturer"/><xsl:text>
   </xsl:text><xsl:value-of select="../Product"/>
 </a></h3>
 <p><xsl:value-of select="."/></p>
</xsl:for-each>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 5.23 A Stylesheet to Demonstrate the generate-id ( ) Function (HTMLEditors.xsl).

```
<html>
<head>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Using the generate-id() function to link software titles to
  descriptions.</title>
</head>
<body>
 <h2>HTML Editors</h2>
  <h3>Allaire Corporation: </h3>
  <p>HomeSite - 4.5</p>
  <p>For further information on HomeSite click here: <a
    href="#b1ab1b7">HomeSite</a></p>
  <h3>Macromedia: </h3>
  <p>DreamWeaver - 4</p>
  <p>For further information on DreamWeaver click here: <a
    href="#b3b7">DreamWeaver</a></p>
  <h3>Microsoft Corporation: </h3>
  <p>FrontPage - 2000</p>
  <p>For further information on FrontPage click here: <a
    href="b7b7">FrontPage</a></p>
  <h3>NetObjects: </h3><p>NetObjects Fusion - 5</p>
  <p>For further information on NetObjects Fusion click here: <a
    href="#b5b7">NetObjects Fusion</a></p>
<h3><a name="b1ab1b7">Allaire Corporation HomeSite</a></h3>
<p>This is a placeholder for comments about Allaire HomeSite version
  4.5.</p>
<h3><a name="b1ab3b7">Macromedia DreamWeaver</a></h3>
<p>This is a placeholder for comments about Macromedia Dreamweaver
  version 4.</p>
<h3><a name="b1ab5b7">NetObjects NetObjects Fusion</a></h3>
<p>This is a placeholder for comments about NetObjects Fusion version
  5.</p>
<h3><a name="b1ab7b7">Microsoft Corporation FrontPage</a></h3>
<p>This is a placeholder for comments about Microsoft FrontPage 2000.</p>
</body>
</html>
```

Listing 5.24 The Output of the Stylesheet in Listing 5.23 (HTMLEditors.html).

When you click on one of the links in the list of products in the first part of the HTML Web page, you are linked to the appropriate <a> element, which contains the content of the <Comments> element for that product.

XSLT key ( ) Function

The key ( ) function of XSLT takes two arguments. The first argument is either a string or is converted to a string using the rules of the string ( ) function. The second argument

may be of any data type and specifies the required value of the key. It returns a node set with the desired key values.

The key ( ) function is designed to work with the XSLT <xsl:key> element. The <xsl:key> element is a top-level element in an XSLT stylesheet, that is, the <xsl:key> element is a child of the <xsl:stylesheet> or <xsl:transform> element. A stylesheet may contain multiple <xsl:key> elements. If implemented in the XSLT processor using, say, an index or hash table, the <xsl:key> element may speed up processing.

The <xsl:key> element takes the following form, with three mandatory attributes:

```
<xsl:key name="something" match="something" use="something"/>
```

The name attribute specifies the name of the key. The value of the name attribute must be a QName. If the QName contains a namespace prefix, then the namespace URI with which the namespace prefix is associated must be in scope on the <xsl:key> element.

The value of the match attribute must be an XSLT pattern. It defines the nodes to which the key applies. The match attribute of the <xsl:key> element is one of the four places where XSLT patterns are used. The other three are the match attribute of the <xsl:template> element and the count and from attributes of the <xsl:number> element.

The value of the use attribute is an expression, which is used to arrive at the value of the key for each of the nodes specified in the match attribute.

NOTE **The <xsl:key> element is always an empty element.**

If we had a simple parts list like that in Listing 5.25, then we could define a key, using the <xsl:key> element like this:

```
<xsl:key name="PartsNumber" match="Part" use="Number"/>
```

```
<PartsList>
<Part>
<Number>ABC123</Number>
</Part>
<Part>
<Number>ABC234</Number>
</Part>
<Part>
<Number>BCD345</Number>
</Part>
<Part>
<Number>CDE456</Number>
</Part>
</PartsList>
```

Listing 5.25 A Short Parts List Expressed in XML (PartsList.xml).

The key ( ) function exists to help find nodes when we know the value of their content. For example, if we wished to use the sample document in Listing 5.26 and the key ( ) function to find books by R. Allen Wyke, given the element

```
<xsl:key name="Author" match="Book" use="Authors/Author"/>
```

we could process books using the key ( ) function to determine the value of a select attribute. For example, the following <xsl:for-each> element would result in each book of which R. Allen Wyke is an author being processed as indicated by the content of the <xsl:for-each> element.

```
<xsl:for-each select="key('Author', 'R. Allen Wyke')"/>
```

```
<?xml version='1.0'?>
<BookCatalog>
<Book Price="44.99">
<Title>XSL Essentials</Title>
<Authors>
  <Author>Michael Fitzgerald</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XHTML Essentials</Title>
<Authors>
  <Author>Michael Sauers</Author>
  <Author>R. Allen Wyke</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XPath Essentials</Title>
<Authors>
  <Author>Andrew Watt</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XML Schema Essentials</Title>
<Authors>
  <Author>Andrew Watt</Author>
  <Author>R. Allen Wyke</Author>
</Authors>
</Book>
</BookCatalog>
```

Listing 5.26 A Short Book Catalog Expressed in XML (BookCatalog2.xml).

XSLT system-property ( ) Function

The XSLT system-property ( ) function returns information about the processing environment.

XSLT unparsed-entity-uri ( ) Function

The XSLT unparsed-entity-uri ( ) function provides access to declarations of unparsed entities in the DTD for the source XML document.

Looking Ahead

Now that we have looked at the functions that can be used with XPath, Chapter 6 will discuss how we can use XPath and XSLT to create XML. Further examples of XPath functions will be discussed in Chapter 13, "Working with XPath Functions."

TEAMFLY

Using XPath and XSLT to Produce XML

This chapter will introduce you to the use of XPath with the Extensible Stylesheet Language Transformations (XSLT) in order to restructure XML from a source document into a result XML document.

Most of the XPath that will be used in this chapter has already been introduced; therefore, most of this chapter will show you code with relatively little explanation, other than that needed for you to understand the purpose of the code. I encourage you to read the code and work out how to use it for yourself. This is an essential step in moving toward writing your own code from scratch.

One of the most common uses for XPath and XSLT in a business to business (B2B) context is for the restructuring of XML. So, later in the chapter, we will look at how we create new elements or attributes. But first, although it may seem a strange way to start a chapter on XPath for producing new structures in XML, we will look at what XPath *cannot* do.

What XPath Can't Do

The XPath data model, as you may remember from Chapter 3, does not include any information on either the XML declaration or about a DOCTYPE declaration. Thus, we have no way, using XPath alone, to define whether or not an XML declaration or a DOCTYPE declaration should be present in the output document or influence what they contain.

The <xsl:output> Element

The solution to these gaps in XPath is the XSLT 1.0 <xsl:output> element. The <xsl:output> element uses XML output as the default setting, although you can also explicitly set that by giving the method attribute a value of "xml." Note, however, that if you want to create XHTML, you need to explicitly set the method attribute to a value of "xml", since the <html> document element in the output document will, otherwise, likely cause an XSLT processor to default to HTML output.

To choose whether or not to include an XML declaration in the output XML document, you can specify the value of the omit-xml-declaration attribute on the <xsl:output> element to be either "yes" or "no". If you choose to have an XML declaration, then you can set the version attribute of the <xsl:output> element. Currently the only legal value is "1.0". Similarly you have the option to set the standalone attribute of <xsl:output> to "yes" or "no" to determine the value of the corresponding attribute on the XML declaration of the output document. The encoding attribute of <xsl:output> allows the encoding attribute of the XML declaration of the output document to be defined.

Let's take the very simple source document in Listing 6.1 and add an XML declaration to it, while otherwise simply copying the source document to the output document.

Listing 6.2 is the XSLT stylesheet that causes an XML declaration to be added to the output document. The values assigned to the version, standalone, and encoding attributes will be obvious from those included in Listing 6.2.

The output from the transformation is shown in Figure 6.1, where you can see that an XML declaration has been created with version, standalone, and encoding attributes.

```
<!-- This is the XML source document which has no XML declaration
or any DOCTYPE declaration. -->
<SimpleDoc>
<Content>
Some content would go here.
</Content>
</SimpleDoc>
```

Listing 6.1 A Simple XML Source Document (SimpleDoc.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
   omit-xml-declaration="no"
   version="1.0"
   standalone="yes"
```

Listing 6.2 A Stylesheet to Add an XML Declaration to the Output Document (SimpleDoc01.xsl).

```
   encoding="ISO-8859-1"
   />

<xsl:template match="/">
<xsl:copy-of select="/SimpleDoc"/>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.2 (*Continued*)

Notice that our XSLT stylesheet did not copy the comment in the source document since it was outside the document element to which we applied the <xsl:copy-of> element. Also it would serve no useful purpose since its content is no longer accurate in the output document. However, the code to copy across such a comment is shown in Listing 6.3.

The <xsl:output> element also allows us to add a DOCTYPE declaration to the output document but does so by a fairly indirect way. The <xsl:output> element has two attributes, the doctype-public attribute and the doctype-system attribute, which define the public and system identifiers to be used in the DOCTYPE declaration in the output document.

In Listing 6.4 you can see a development of the previous example that allows us to add a DOCTYPE declaration to the output document. Notice that I have used highly unusual public and system identifiers to demonstrate that at their simplest, you are simply outputting strings.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
   omit-xml-declaration="no"
   method="xml" media-type="text/html"
   version="1.0"
   standalone="yes"
   encoding="ISO-8859-1"
   />

<xsl:template match="/">
<xsl:comment>
<xsl:value-of select="/child::comment()"/>
</xsl:comment>
<xsl:copy-of select="/SimpleDoc"/>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.3 A Stylesheet to Output a Comment Outside the Source Document Element Root (SimpleDoc01a.xsl).

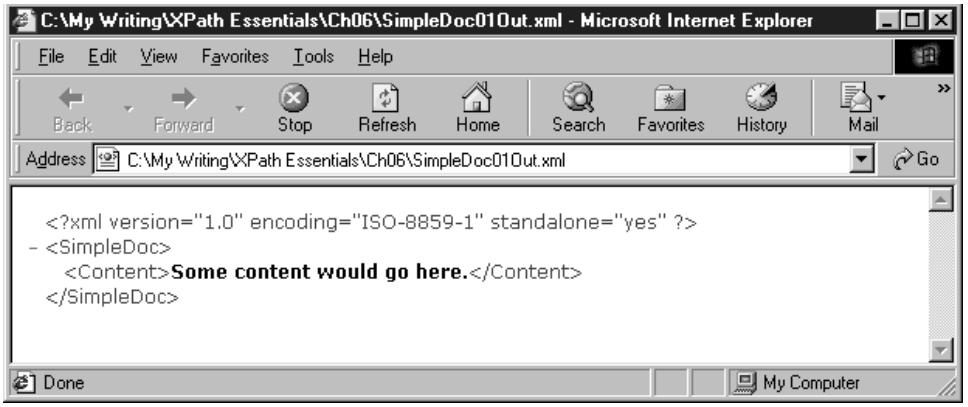


Figure 6.1 Adding an XML Declaration and Its Attributes to an Output XML Document.

The output from the transformation is shown in Listing 6.5. Notice that the XSLT processor has taken the strings that we supplied in doctype-public and doctype-system attributes of the <xsl:output> element in Listing 6.4 and output them without change. The point I want to emphasize is that there is no error checking as to whether or not the public or system identifiers are correct or make any sense. As far as an XSLT processor is concerned, they are both simply strings. It is up to you to make sure you get the syntax correct.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
   omit-xml-declaration="no"
   version="1.0"
   standalone="yes"
   encoding="ISO-8859-1"
   doctype-public="Mary had a little lamb"
   doctype-system="Its fleece was white as snow"
   />

<xsl:template match="/">
<xsl:copy-of select="/SimpleDoc"/>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.4 A Stylesheet to Add a DOCTYPE Declaration to the Output Document (SimpleDoc02.xsl).

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!DOCTYPE SimpleDoc
  PUBLIC "Mary had a little lamb"
  "Its fleece was white as snow">
<SimpleDoc>
<Content>
Some content would go here.
</Content>
</SimpleDoc>
```

Listing 6.5 An XML Document Output by Listing 6.4 (SimpleDoc02Out.xml).

Notice that the XSLT processor automatically supplies within the DOCTYPE declaration the element type name of the document element. The content of the public and system identifiers is your responsibility.

If we create an XHTML document, which is, after all, an application language of XML, to display the content of SimpleDoc.xml, we need to provide accurate public and system identifiers if we want to be sure that it will be handled correctly by a browser or other processor. If we assume that we want to use the DOCTYPE declaration for XHTML 1.0 Transitional, we can do so using Listing 6.6.

The output XHTML 1.0 Transitional document is shown in Listing 6.7. Notice that I have explicitly included on the <xsl:output> element a method attribute with value of "xml" in order to ensure that the XSLT processor outputs as XML, not as HTML. For the earlier examples we could make use of the default setting of XML, but when an XSLT processor meets an output document that has a document element of <html>, it tends

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
   omit-xml-declaration="no"
   method="xml" media-type="text/html"
   version="1.0"
   standalone="yes"
   encoding="UTF-8"
   doctype-public="-//W3C//DTD XHTML 1.0 Transitional//EN"
   doctype-system="DTD/xhtml1-transitional.dtd"
   indent="yes"
   />

<xsl:template match="/">
<html>
<head>
<title>A simple XHTML 1.0 Document with Doctype declaration.</title>
```

Listing 6.6 A Stylesheet to Output an XHTML 1.0 Output Document (SimpleDoc03.xsl).
(*continues*)

```
</head>
<body>
<p>The content of the &lt;Content&gt; element is: <xsl:value-of
select="/SimpleDoc/Content"/></p>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.6 (*Continues*)

to assume that you want HTML output unless you make it explicit that output should be XML.

If we want the output document to be a particular flavor of XML, for example, Scalable Vector Graphics (SVG), which we will consider in more detail in Chapter 8, "Using XPath and XSLT to Produce SVG," then we can make use of the media-type attribute on the <xsl:output> element. For example, if we wanted to create an SVG output document, the <xsl:output> element would look like this (which includes the system identifier for the SVG Proposed Recommendation of July 19, 2001):

```
<xsl:output
   omit-xml-declaration="no"
   version="1.0"
   standalone="yes"
   encoding="UTF-8"
   doctype-public"-//W3C//DTD SVG 1.0//EN"
```

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<!DOCTYPE html
  PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "DTD/xhtml1-
    transitional.dtd">

<html>
   <head>
      <title>A simple XHTML 1.0 Document with Doctype declaration.</title>
   </head>
   <body>
      <p>The content of the &lt;Content&gt; element is:
Some content would go here.
</p>
   </body>
</html>
```

Listing 6.7 The XHTML Document Output by Listing 6.6 (SimpleDoc03.html).

```
   doctype-system="http://www.w3.org/TR/2001/PR-SVG-
20010719/DTD/svg10.dtd">"
   indent="yes"
   media-type="image/svg+xml"
   />
```

Creating Elements

To create elements or attributes is a very basic task when converting from one XML structure to another. First we will look at the creation of new XML elements, which can be of generic XML as in the examples in this chapter, or in a specific application language of XML such as SVG, which you will see created in Chapter 8, "Using XPath and XSLT to produce SVG."

Selecting and Creating Elements

In order to transform one XML document to another, it is fundamental that you select element nodes from the source tree and create element nodes in the result tree. We can use XPath with several XSLT elements to make such choices and/or create new elements.

First, let's look at how you can use the <xsl:element> element to create new elements in the output document.

Using the <xsl:element> Element

The characteristics of the <xsl:element> element were summarized in Chapter 1.

Let's take an example of a situation where we want to create a simple summary of a purchase. The purchase data is shown in Listing 6.8.

In the stylesheet that follows (see Listing 6.9), we use <xsl:element> several times to create new elements in the output document, SimplePurchaseOut.xml (see Listing 6.10).

What we want to do is to create a new element called <PurchaseSummary>, which indicates how many items have been purchased in order and which will be created with an attribute called PurchaserID (which is derived from the <PurchaserID> element in the source document). Additionally, we will restructure the document so that several

```
<?xml version='1.0'?>
<SimplePurchase>
<CustomerID>9443</CustomerID>
<Date>2002-03-30</Date>
<Items>
<Item SKU="DD9981" Quantity="4" Price="3.00">Dooda</Item>
<Item SKU="WD1239" Quantity="2" Price="10.00">Widget</Item>
<Item SKU="ABC3882" Quantity="1" Price="220.00">Acme Fitsat</Item>
</Items>
</SimplePurchase>
```

Listing 6.8 A Simple Purchase Represented in XML (SimplePurchase.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<xsl:element name="PurchaseSummary">
<xsl:attribute name="PurchaserID">
<xsl:value-of select="/SimplePurchase/CustomerID"/>
</xsl:attribute>
<xsl:element name="NumItemsBought">
<xsl:value-of select="count(/SimplePurchase/Items/Item)"/>
</xsl:element>
<xsl:apply-templates select="/SimplePurchase/Items/Item"/>
</xsl:element>
</xsl:template>

<xsl:template match="Item">
<xsl:element name="Item">
<xsl:value-of select="."/>
</xsl:element>
<xsl:element name="SKU">
<xsl:value-of select="@SKU"/>
</xsl:element>
<xsl:element name="Quantity">
<xsl:value-of select="@Quantity"/>
</xsl:element>
<xsl:element name="Price">
<xsl:value-of select="@Price"/>
</xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.9 A Stylesheet to Create New Elements Using the <xsl:element> Element (SimplePurchase.xsl).

attributes in the source document are now created as new elements named <SKU>, <Quantity>, and <Price>.

The XML document that we created using the XSLT transformation is shown in Listing 6.10.

We have used the <xsl:element> element where we could have used literal result elements, but by using the <xsl:element> element in this way, we can make greater use of the information that is already available in the source document, assuming that it meets our needs for the structure of the output document.

We can also use the <xsl:element> to create new elements with a namespace prefix and URI. To do that we need to add a namespace attribute to each <xsl:element> that we wish to be associated with the namespace. If, for example, we wished to add a namespace to the output document, we could do so using the stylesheet in Listing 6.11. Each

```
<?xml version="1.0" encoding="utf-8"?>
<PurchaseSummary PurchaserID="9443">
   <NumItemsBought>3</NumItemsBought>
   <Item>Dooda</Item>
   <SKU>DD9981</SKU>
   <Quantity>4</Quantity>
   <Price>3.00</Price>
   <Item>Widget</Item>
   <SKU>WD1239</SKU>
   <Quantity>2</Quantity>
   <Price>10.00</Price>
   <Item>Acme Fitsat</Item>
   <SKU>ABC3882</SKU>
   <Quantity>1</Quantity>
   <Price>220.00</Price>
</PurchaseSummary>
```

Listing 6.10 The XML Document Output by Listing 6.9 (SimplePurchaseOut.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
   method="xml"
   indent="yes"
   />

<xsl:template match="/">
<xsl:element name="XMML:PurchaseSummary"
namespace="http://www.XMML.com/Finance">
<xsl:attribute name="PurchaserID">
<xsl:value-of select="/SimplePurchase/CustomerID"/>
</xsl:attribute>
<xsl:element name="XMML:NumItemsBought"
namespace="http://www.XMML.com/Finance">
<xsl:value-of select="count(/SimplePurchase/Items/Item)"/>
</xsl:element>
<xsl:apply-templates select="/SimplePurchase/Items/Item"/>
</xsl:element>
</xsl:template>

<xsl:template match="Item">
<xsl:element name="XMML:Item" namespace="http://www.XMML.com/Finance">
<xsl:value-of select="."/>
<xsl:element name="XMML:SKU" namespace="http://www.XMML.com/Finance">
```

Listing 6.11 A Stylesheet to Create New Elements That Have a Namespace Prefix (SimplePurchase02.xsl). (*continues*)

```
<xsl:value-of select="@SKU"/>
</xsl:element>
<xsl:element name="XMML:Quantity"
namespace="http://www.XMML.com/Finance">
<xsl:value-of select="@Quantity"/>
</xsl:element>
<xsl:element name="XMML:Price" namespace="http://www.XMML.com/Finance">
<xsl:value-of select="@Price"/>
</xsl:element>
</xsl:element>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.11 (*Continued*)

of the elements created in the output document (Listing 6.12) has an XMML namespace prefix associated with the namespace URI "http://www.XMML.com/Finance".

The output XML document with namespace prefixes in place is shown in Listing 6.12.

Having looked at how we can use <xsl:element> to create new elements, let's move on and look at how we can select and copy across elements from an XML source document.

Using the <xsl:copy> Element

The <xsl:copy> element is described in the latter part of Chapter 1. It is useful when you want to carry out a "shallow copy"—to copy a node without copying any of its descendant or attribute nodes. For example, if we had the source document shown in Listing 6.13, we can

TEAM-FLY

```
<?xml version="1.0" encoding="utf-8"?>
<XMML:PurchaseSummary xmlns:XMML="http://www.XMML.com/Finance"
PurchaserID="9443">
   <XMML:NumItemsBought>3</XMML:NumItemsBought>
   <XMML:Item>Dooda<XMML:SKU>DD9981</XMML:SKU>
      <XMML:Quantity>4</XMML:Quantity>
      <XMML:Price>3.00</XMML:Price>
   </XMML:Item>
   <XMML:Item>Widget<XMML:SKU>WD1239</XMML:SKU>
      <XMML:Quantity>2</XMML:Quantity>
      <XMML:Price>10.00</XMML:Price>
   </XMML:Item>
   <XMML:Item>Acme Fitsat<XMML:SKU>ABC3882</XMML:SKU>
      <XMML:Quantity>1</XMML:Quantity>
      <XMML:Price>220.00</XMML:Price>
   </XMML:Item>
</XMML:PurchaseSummary>
```

Listing 6.12 The XML Document Output by Listing 6.11 (SimplePurchase02Out.xml).

```
<?xml version='1.0'?>
<RegisteredUsers>
<User>
<Name>
<FirstName>Jonathan</FirstName>
<MiddleInitial>F</MiddleInitial>
<LastName>Klingon</LastName>
</Name>
<EmailAddress>JFK@Omega99.com</EmailAddress>
<UserNumber>PLE8870</UserNumber>
<Nickname>Pres</Nickname>
</User>
<User>
<Name>
<FirstName>Patrick</FirstName>
<MiddleInitial>T</MiddleInitial>
<LastName>Kirk</LastName>
</Name>
<EmailAddress>PTK@Galafrey.com</EmailAddress>
<UserNumber>K9FOOT</UserNumber>
<Nickname>Paw</Nickname>
</User>
</RegisteredUsers>
```

Listing 6.13 A Brief Database of Registered Users Expressed in XML (RegisteredUsers.xml).

create some new elements using <xsl:element> but also use <xsl:copy> to copy some elements from the source document.

We can use the stylesheet shown in Listing 6.14 to produce a personalized message for registered users whose user number begins with "K9".

The output displayed on screen is shown in Figure 6.2.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
   method="xml"
   indent="yes"
   omit-xml-declaration="no"
   />
<xsl:preserve-space elements="*"/>

<xsl:template match="/">
```

Listing 6.14 A Stylesheet Using <xsl:copy> to Create an Element in the Output Document (RegisteredUsers.xsl). (*continues*)

```
<xsl:element name="Welcome">
<xsl:apply-templates select="/RegisteredUsers/User"/>
</xsl:element>
</xsl:template>

<xsl:template match="User">
<xsl:if test="starts-with(UserNumber, 'K9')">
<xsl:value-of select="Name/FirstName"/>, welcome to XMML.com. We will
  use the email address, <xsl:value-of select="EmailAddress"/> to
  communicate with you at all times.<xsl:text>

Please note the following information for future reference:</xsl:text>
<xsl:apply-templates select="UserNumber"/>
<xsl:apply-templates select="Nickname"/>
</xsl:if>
</xsl:template>

<xsl:template match="UserNumber | Nickname">
<xsl:copy>
<xsl:value-of select="."/>
</xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.14 (*Continued*)

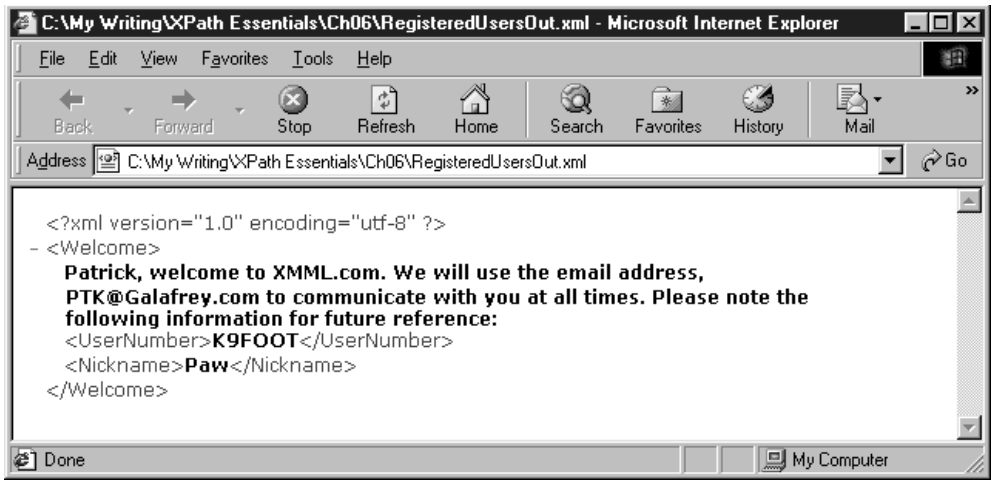


Figure 6.2 Using xsl:copy to Copy Elements to an Output Document.

Using the <xsl:copy-of> Element

In the previous example when we used the <xsl:copy> element, it was necessary to insert the `<xsl:value-of select="."/>` to copy across the element's content. If we had not done that, an empty element would have been created in the output document. The <xsl:copy-of> element allows us to create a "deep copy" (that is, when we copy across an element node we also automatically copy across its child nodes and descendant nodes as well as any associated namespace and attribute nodes).

Let's suppose that when creating the message for registered users we wanted to ensure that all information held about a person was made known to each of them. We could use the stylesheet shown in Listing 6.15 to achieve that, using the <xsl:copy-of> element.

The <xsl:copy-of> element uses the select attribute to define which node is to copied using a deep copy.

The document created is shown in Figure 6.3.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
   method="xml"
   indent="yes"
   omit-xml-declaration="no"
   />
<xsl:preserve-space elements="*"/>

<xsl:template match="/">
<xsl:element name="Welcome">
<xsl:apply-templates select="/RegisteredUsers/User"/>
</xsl:element>
</xsl:template>

<xsl:template match="User">
<xsl:if test="starts-with(UserNumber, 'K9')">
<xsl:value-of select="Name/FirstName"/>, welcome to XMML.com. We will
  use
the email address, <xsl:value-of select="EmailAddress"/> to communicate
  with you at all times.<xsl:text>

Please note the following information for future reference:</xsl:text>
<xsl:apply-templates select="UserNumber"/>
<xsl:apply-templates select="Nickname"/>
<xsl:text>The full information which we hold about you is listed below.
  Please feel free to correct any errors or omissions.</xsl:text>
<xsl:copy-of select="."/>
</xsl:if>
```

Listing 6.15 A Stylesheet to Create an Element in the Output Document Using <xsl:copy> (RegisteredUsers02.xsl). (*continues*)

```
</xsl:template>

<xsl:template match="UserNumber | Nickname">
<xsl:copy>
<xsl:value-of select="."/>
</xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.15 (*Continued*)

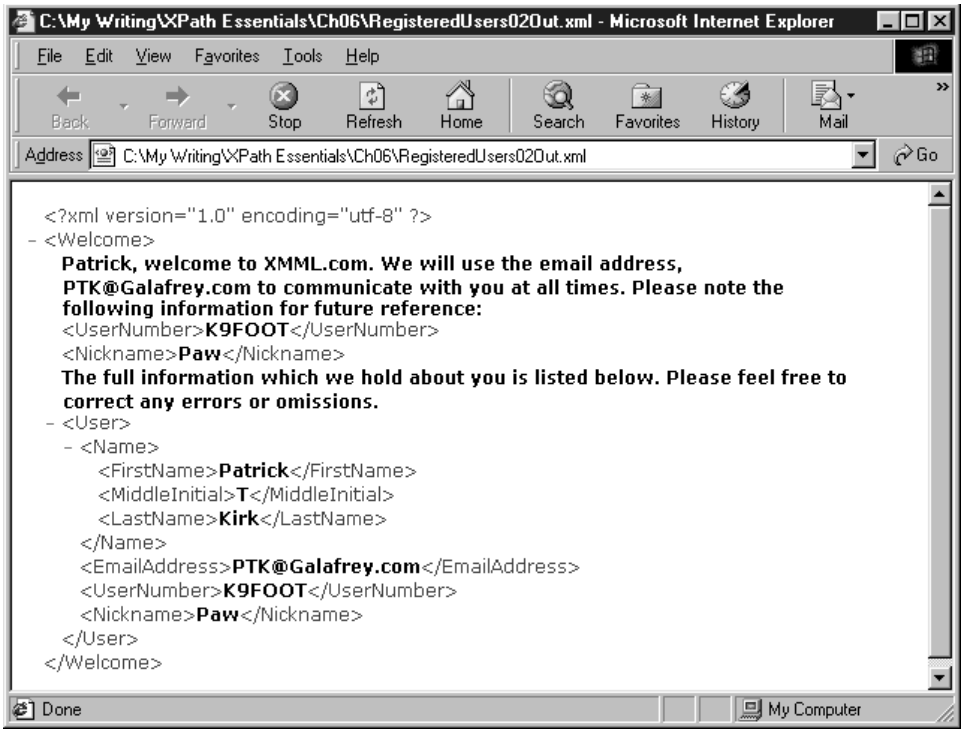


Figure 6.3 Using the <xsl:copy-of> Element to Deep Copy an Element Node in the Source Document.

As well as creating new code, XPath and XSLT will often be used to reorder elements from one XML document to another.

Reordering Content

Whether you need to use XPath and XSLT to reorder content of your XML documents will depend, at least in part, on whether you hold any data as XML per se, or whether

your data is stored in a relational database that can output data as XML. Let's assume that you do store data as XML and want to select and sort some data.

If we had a simple source document such as that in Listing 6.16, we would likely want to have it sorted in some fashion, whereas currently it seems to be totally unsorted.

Let's suppose that a report was needed that required that the orders be sorted by date and secondarily by customer name. We can create this report using the stylesheet shown in Listing 6.17.

The sorted XML document OrdersSortedOut.xml is shown in Listing 6.18.

```
<?xml version='1.0'?>
<Orders>
<Order>
<Date>2001-07-28</Date>
<Customer>Schmidt Enterprises</Customer>
<OrderReference>SE903</OrderReference>
</Order>
<Order>
<Date>2002-09-08</Date>
<Customer>SVGenius.com</Customer>
<OrderReference>SVG134</OrderReference>
</Order>
<Order>
<Date>2001-04-29</Date>
<Customer>ECXML.com</Customer>
<OrderReference>ECX004</OrderReference>
</Order>
<Order>
<Date>2002-03-7</Date>
<Customer>Wiley</Customer>
<OrderReference>WIL123</OrderReference>
</Order>
<Order>
<Date>2001-07-28</Date>
<Customer>Curly Pow</Customer>
<OrderReference>CUR777</OrderReference>
</Order>
</Orders>
```

Listing 6.16 A Short Orders Database Expressed in XML (Orders.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Listing 6.17 A Stylesheet to Order Elements in the Source Document (OrdersSorted.xsl). (*continues*)

```
<xsl:output
  method="xml"
  indent="yes"
  />

<xsl:template match="/">
<xsl:element name="Orders">
<xsl:call-template name="SortByDate"/>
</xsl:element>
</xsl:template>

<xsl:template name="SortByDate">
<xsl:for-each select="Orders/Order">
<xsl:sort select="Date"/>
<xsl:sort select="Customer"/>
<xsl:copy-of select="."/>
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.17 (*Continued*)

```
<?xml version="1.0" encoding="utf-8"?>
<Orders>
   <Order>
      <Date>2001-04-29</Date>
      <Customer>ECXML.com</Customer>
      <OrderReference>ECX004</OrderReference>
   </Order>
   <Order>
      <Date>2001-07-28</Date>
      <Customer>Curly Pow</Customer>
      <OrderReference>CUR777</OrderReference>
   </Order>
   <Order>
      <Date>2001-07-28</Date>
      <Customer>Schmidt Enterprises</Customer>
      <OrderReference>SE903</OrderReference>
   </Order>
   <Order>
      <Date>2002-03-07</Date>
      <Customer>Wiley</Customer>
      <OrderReference>WIL123</OrderReference>
```

Listing 6.18 The Output of the Stylesheet in Listing 6.17 (OrdersSortedOut.xml).

```
   </Order>
   <Order>
      <Date>2002-09-08</Date>
      <Customer>SVGenius.com</Customer>
      <OrderReference>SVG134</OrderReference>
   </Order>
</Orders>
```

Listing 6.18 (*Continued*)

Reusing Business Information

A basic principle of data management is to minimize the number of times a piece of data is input and, as far as possible, ensure that a single piece of data is stored in only one location, thereby minimizing the likelihood of inconsistencies in data and facilitating data maintenance. Thus, when we receive data from a business partner that is already in XML, it makes a lot of sense to reuse as much of that information as possible.

A basic business document is the purchase order and its related document—the invoice. So let's examine how we can use the information contained in an XML purchase order in producing an XML-based invoice.

In Listing 6.19 you see a fairly generic purchase order. In this example scenario, the purchase order is sent to XMML.com, which then wants to be able to use the information to generate an invoice for the customer.

```
<?xml version='1.0'?>
<PurchaseOrder number="2002/A893">
<From>A. Customer</From>
<Contact phone="(123) 456 7890">Fred Schmidt</Contact>
<Delivery>
<Address1>23456 Fifth Avenue</Address1>
<Address2></Address2>
<City>New York</City>
<PostalCode>00002</PostalCode>
<Country>USA</Country>
</Delivery>
<Billing>
<BillingContact>Peter Bolshoi</BillingContact>
<BillingAddress>
<Address1>23456 Fifth Avenue</Address1>
<Address2></Address2>
<City>New York</City>
<PostalCode>00002</PostalCode>
```

Listing 6.19 A Simple Purchase Order Expressed in XML (PurchaseOrder.xml).

(*continues*)

```
<Country>USA</Country>
</BillingAddress>
</Billing>
<Items>
<Item>
<PartNumber>XMML123</PartNumber>
<Description>On-Site Training</Description>
<Quantity>4</Quantity>
<Comment>To be carried out on July 8th and 9th at our New York site, as
  discussed.</Comment>
</Item>
<Item>
<PartNumber>XMML999</PartNumber>
<Description>XSLT Consultancy</Description>
<Quantity>1</Quantity>
<Comment>Re the implementation of a new Web strategy, as described in
  our letter of intent dated
May 30th.</Comment>
</Item>
</Items>
</PurchaseOrder>
```

Listing 6.19 (*Continued*)

In this example purchase order, I have shown the use of the default namespace but, in real life, it would make sense for both business entities to use XML namespaces to avoid or minimize the likelihood of element type name clashes. The invoice to be generated by XMML.com will use a specific namespace prefix and namespace URI. The XPath expressions shown as being used to select nodes from the purchase order would likely also possess namespace prefixes. For simplicity those are omitted in Listing 6.20.

And the output invoice in XML format is shown in Listing 6.21.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:XMML="http://www.xmml.com/Finance/"
                >

<xsl:output
   method="xml"
   omit-xml-declaration="no"
   version="1.0"
   encoding="ISO-8859-1"
   indent="yes"
```

Listing 6.20 A Stylesheet to Generate an Invoice from a Customer's Purchase Order (PurchaseOrder.xsl).

```
   />

<xsl:template match="/">
<XMML:Invoice>
<xsl:call-template name="CreateHeader"/>
<xsl:call-template name="BillTo"/>
<xsl:call-template name="ItemsBilled"/>
</XMML:Invoice>
</xsl:template>

<xsl:template name="CreateHeader">
<XMML:BillingFrom>
<XMML:From>XMML.com</XMML:From>
<XMML:Contact>Billing@XMML.com</XMML:Contact>
<XMML:BillingType>Training</XMML:BillingType>
</XMML:BillingFrom>
</xsl:template>

<xsl:template name="BillTo">
<XMML:BillingTo>
<XMML:BillingCompany><xsl:value-of select="/PurchaseOrder/From"/>
</XMML:BillingCompany>
<XMML:FAO><xsl:value-of select="PurchaseOrder/Billing/BillingContact"/>
</XMML:FAO>
<XMML:BillingAddress1><xsl:value-of
select="/PurchaseOrder/Billing/BillingAddress/Address1"/></XMML:
  BillingAddress1>

<xsl:if
test="/PurchaseOrder/Billing/BillingAddress/Address2/child::text()">
<XMML:BillingAddress2><xsl:value-of
select="/PurchaseOrder/Billing/BillingAddress/Address2"/></XMML:
  BillingAddress2>
</xsl:if>
<XMML:BillingCity><xsl:value-of
select="/PurchaseOrder/Billing/BillingAddress/City"/></XMML:BillingCity>
<XMML:BillingPostalCode><xsl:value-of select="/PurchaseOrder/Billing/
  BillingAddress/PostalCode"/></XMML:BillingPostalCode>
<XMML:BillingCountry><xsl:value-of
select="/PurchaseOrder/Billing/BillingAddress/Country"/></XMML:
  BillingCountry>
</XMML:BillingTo>
</xsl:template>

<xsl:template name="ItemsBilled">
<XMML:ItemsBilled>
<xsl:for-each select="/PurchaseOrder/Items/Item">
<xsl:if test="starts-with(PartNumber, 'XMML1')">
```

Listing 6.20 (Continued) (*continues*)

```
<XMML:Code><xsl:value-of
select="/PurchaseOrder/Items/Item/PartNumber"/></XMML:Code>
<XMML:Description><xsl:value-of
select="/PurchaseOrder/Items/Item/Description"/></XMML:Description>
<XMML:CustomerRequest><xsl:value-of
select="/PurchaseOrder/Items/Item/Comment"/></XMML:CustomerRequest>
</xsl:if>
</xsl:for-each>
</XMML:ItemsBilled>
</xsl:template>

</xsl:stylesheet>
```

Listing 6.20 (*Continued*)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<XMML:Invoice xmlns:XMML="http://www.xmml.com/Finance/">
   <XMML:BillingFrom>
      <XMML:From>XMML.com</XMML:From>
      <XMML:Contact>Billing@XMML.com</XMML:Contact>
      <XMML:BillingType>Training</XMML:BillingType>
   </XMML:BillingFrom>
   <XMML:BillingTo>
      <XMML:BillingCompany>A. Customer</XMML:BillingCompany>
      <XMML:FAO>Peter Bolshoi</XMML:FAO>
      <XMML:BillingAddress1>23456 Fifth Avenue</XMML:BillingAddress1>
      <XMML:BillingCity>New York</XMML:BillingCity>
      <XMML:BillingPostalCode>00002</XMML:BillingPostalCode>
      <XMML:BillingCountry>USA</XMML:BillingCountry>
   </XMML:BillingTo>
   <XMML:ItemsBilled>
      <XMML:Code>XMML123</XMML:Code>
      <XMML:Description>On-Site Training</XMML:Description>
      <XMML:CustomerRequest>To be carried out on July 8th and 9th at
        our New York site, as discussed.</XMML:CustomerRequest>
   </XMML:ItemsBilled>
</XMML:Invoice>
```

Listing 6.21 The Output of the Stylesheet Shown in Listing 6.20 (XMMLInvoice.xml).

Creating Attributes

I have shown you how to use XPath and XSLT to create elements, either to contain new information or to contain information held in attributes in the source XML document. Of course, the opposite type of transformation also needs to be carried out frequently—creating attributes to contain information previously held as elements.

Listing 6.18 (repeated in Listing 6.22 for convenience) is the source XML document for an example that shows how to create a new XML document that has attributes where elements were present in the source document. Creating new attributes makes use of the <xsl:attribute> element, which was described in Chapter 1.

The stylesheet used to created new attributes using elements in the source document is shown in Listing 6.23.

The output document, with the full complement of newly created attributes, is shown in Listing 6.24.

```
<?xml version="1.0" encoding="utf-8"?>
<Orders>
   <Order>
      <Date>2001-04-29</Date>
      <Customer>ECXML.com</Customer>
      <OrderReference>ECX004</OrderReference>
   </Order>
   <Order>
      <Date>2001-07-28</Date>
      <Customer>Curly Pow</Customer>
      <OrderReference>CUR777</OrderReference>
   </Order>
   <Order>
      <Date>2001-07-28</Date>
      <Customer>Schmidt Enterprises</Customer>
      <OrderReference>SE903</OrderReference>
   </Order>
   <Order>
      <Date>2002-03-07</Date>
      <Customer>Wiley</Customer>
      <OrderReference>WIL123</OrderReference>
   </Order>
   <Order>
      <Date>2002-09-08</Date>
      <Customer>SVGenius.com</Customer>
      <OrderReference>SVG134</OrderReference>
   </Order>
</Orders>
```

Listing 6.22 A Sorted List of Orders Expressed in XML (OrdersSortedOut).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

Listing 6.23 A Stylesheet to Create New Attributes in the Output Document (Orders–SortedOut.xsl) (*continues*)

```
<xsl:output
  method="xml"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<Orders>
<xsl:apply-templates select="/Orders/Order"/>
</Orders>
</xsl:template>

<xsl:template match="Order">
<xsl:copy>
<xsl:for-each select="*">
<xsl:attribute name="{name(.)}">
<xsl:value-of select="."/>
</xsl:attribute>
</xsl:for-each>
</xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

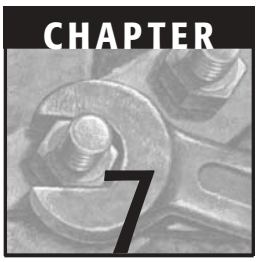
Listing 6.23 (*Continued*)

```
<?xml version="1.0" encoding="utf-8"?>
<Orders>
   <Order Date="2001-04-29" Customer="ECXML.com" OrderReference="ECX004"/>
   <Order Date="2001-07-28" Customer="Curly Pow" OrderReference="CUR777"/>
   <Order Date="2001-07-28" Customer="Schmidt Enterprises"
OrderReference="SE903"/>
   <Order Date="2002-03-07" Customer="Wiley" OrderReference="WIL123"/>
   <Order Date="2002-09-08" Customer="SVGenius.com"
OrderReference="SVG134"/>
</Orders>
```

Listing 6.24 The XML Document Output by Listing 6.22 (OrdersToAttributes.xml).

Looking Ahead

In this chapter we have looked at one of the two most common uses of XPath with XSLT—to restructure XML. In Chapter 7 we will take a closer look at the most common current use of XPath with XSLT—to generate HTML output from XML source documents.



Using XPath and XSLT to Produce HTML

This chapter will introduce and demonstrate the use of XPath with XSLT to create HTML/XHTML. In earlier chapters, we already saw some straightforward examples of using XPath with XSLT to produce HTML.

Using XPath and XSLT to produce HTML or XHTML is one of the most important and common uses of XPath at the present time. This may change in the future, but until stable XML-dedicated browsers are generally available and adopted, there will be a substantial continuing need to convert data held as XML into HTML for display on screen.

A Simple HTML Example

On the assumption that you might have decided to skip some of the earlier chapters, I will start with a very simple example. If we had the XML source document in Listing 7.1

```
<?xml version='1.0'?>
<HelloWorld greeting="First XPath Greeting">
Welcome to the world of using XPath to create HTML!
</HelloWorld>
```

Listing 7.1 A Simple Greeting in XML (HelloWorld.xml).

and wanted to create a simple HTML Web page with the value of the greeting attribute as the content of the HTML <title> element and the content of the <HelloWorld> element to be displayed in the body of the HTML page, we could achieve that using the XSLT stylesheet in Listing 7.2.

Within what I call the "main template"—the <xsl:template> element that matches the root node—we create a skeleton of an HTML document. In the <title> element of what will be the HTML document I have inserted the value of the greeting attribute of the <HelloWorld> element, using the <xsl:value-of> element. The <xsl:apply-templates> element in the main template matches, by default, the element child node of the root node, which is the element node representing the <HelloWorld> element. The <xsl:value-of> element within the applied template then outputs the content of the <HelloWorld> element.

As you can see in Figure 7.1, the value of the greeting attribute is successfully displayed in the title bar of the Web browser, and the content of the <HelloWorld> element is displayed in the body of the HTML page.

In the XSLT stylesheet in Listing 7.2, you saw two very commonly used XPath selections of an attribute and an element in the source document. The following code creates an HTML <title> element, and within it the value of the greeting attribute from the source document is displayed:

```
<title><xsl:value-of select="/HelloWorld/@greeting"/></title>
```

The XPath location path that produces the value of the select attribute of the <xsl:value-of> element indicates that we start at the root node, "/", look for an element

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
<html>
<head>
<title><xsl:value-of select="/HelloWorld/@greeting"/></title>
</head>
<body>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>

<xsl:template match="HelloWorld">
<h3><xsl:value-of select="."/></h3>
<br />
</xsl:template>
</xsl:stylesheet>
```

Listing 7.2 A Stylesheet to Create a Simple HTML Document (HelloWorld.xsl).

First XPath Greeting - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address C:\My Writing\XPath Essentials\Ch07\HelloWorld.html
Go
Welcome to the world of using XPath to create HTML!
Done
My Computer

Figure 7.1 A Simple Transformation to HTML.

child node that represents an element called <HelloWorld>, and then look for an attribute called "greeting" belonging to the <HelloWorld> element. Since such an attribute exists in our source document, its value is copied into the <title> element, which, after the output document has been serialized, is then displayed in the title bar of the browser window.

Similarly, the effect of the <xsl:value-of> element in the following code depends on the context node:

```
<xsl:template match="HelloWorld">
<h3><xsl:value-of select="."/></h3>
<br />
</xsl:template>
```

The <xsl:template> element specifies that the match is on a node representing a <HelloWorld> element. Therefore, that is the context node. The select attribute of the <xsl:value-of> element selects the content of the context node itself and it is its value that is displayed.

You have already seen many simple examples of creating HTML, so in the examples that follow I will look at the creation of a list and a table in HTML and make use of a number of XPath functions.

Creating an HTML List

When you want to display some selected content from an XML source document, an HTML list, whether ordered or unordered, is something you will often want to create. Let's assume we have information about musical composers held as XML, as in Listing 7.3, and we want to display that information in an HTML page and we also want to create, for each composer, a link to a file where further information is available.

The XSLT stylesheet in Listing 7.4 uses the <xsl:apply-templates> element in the main template to select for display only those composers whose surname begins with "D". To make that selection, the XPath starts-with ( ) function is used.

```
<?xml version='1.0'?>
<Composers>
<Composer>
<FirstName>Ludwig</FirstName>
<LastName>Beethoven</LastName>
<Link>Beethoven.xml</Link>
</Composer>
<Composer>
<FirstName>Wolfgang</FirstName>
<LastName>Mozart</LastName>
<Link>Mozart.xml</Link>
</Composer>
<Composer>
<FirstName>Peter</FirstName>
<LastName>Tchaikovsky</LastName>
<Link>Tchaikovsky.xml</Link>
</Composer>
<Composer>
<FirstName>Antonin</FirstName>
<LastName>Dvorak</LastName>
<Link>Dvorak.xml</Link>
</Composer>
<Composer>
<FirstName>Antonio</FirstName>
<LastName>Vivaldi</LastName>
<Link>Vivaldi.xml</Link>
</Composer>
<Composer>
<FirstName>Frederick</FirstName>
<LastName>Delius</LastName>
<Link>Delius.xml</Link>
</Composer>
</Composers>
```

Listing 7.3 A Short Catalog of Composers Expressed in XML (Composers.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"/>
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
```

Listing 7.4 A Stylesheet Using the starts-with ( ) Function (Composers.xsl).

```
<head>
<title><xsl:value-of select="name(/*)"/></title>
</head>
<body>
<p>On this site you can access information about some of the world's
  most famous composers.</p>
<ol>
<xsl:apply-templates select="/Composers/Composer[starts-with(LastName,
  'D')]">
<xsl:sort select="LastName"/>
</xsl:apply-templates>
</ol>
</body>
</html>
</xsl:template>

<xsl:template match="Composer">
<li><a href="{Link}"> <xsl:value-of select="LastName"/>, <xsl:value-of
  select="FirstName"/></a><br /></li>
</xsl:template>

</xsl:stylesheet>
```

Listing 7.4 (*Continued*)

The <xsl:template> element, which matches "Composer", creates a list item within which is nested a simple HTML hyperlink for each composer whose information was selected in the main template. The href attribute of the <a> element created uses an attribute value template to make use of the information in the <Link> element in the source document. Thereafter, the last name and first name of the composer are selected from the source document. The HTML document created is shown in Listing 7.5.

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
      <title>Composers</title>
   </head>
   <body>
      <p>On this site you can access information about some of the
        world's most famous composers.</p>
      <ol>
         <li><a href="Delius.xml">Delius, Frederick</a><br></li>
         <li><a href="Dvorak.xml">Dvorak, Antonin</a><br></li>
      </ol>
   </body>
</html>
```

Listing 7.5 The HTML Document Output by Listing 7.4 (Composers.html).

If we add a little more information to our source document, as in Listing 7.6, we can use that to create a slightly more complex HTML output.

First we can use a stylesheet to order the composers by country and then within that sort we can sort the composers by last name, as shown in Listing 7.7.

```
<?xml version='1.0'?>
<Composers>
<Composer>
<FirstName>Ludwig</FirstName>
<LastName>Beethoven</LastName>
<Country>Germany</Country>
<Link>Beethoven.xml</Link>
</Composer>
<Composer>
<FirstName>Wolfgang</FirstName>
<LastName>Mozart</LastName>
<Country>Austria</Country>
<Link>Mozart.xml</Link>
</Composer>
<Composer>
<FirstName>Peter</FirstName>
<LastName>Tchaikovsky</LastName>
<Country>Russia</Country>
<Link>Tchaikovsky.xml</Link>
</Composer>
<Composer>
<FirstName>Antonin</FirstName>
<LastName>Dvorak</LastName>
<Country>Czechoslovakia</Country>
<Link>Dvorak.xml</Link>
</Composer>
<Composer>
<FirstName>Antonio</FirstName>
<LastName>Vivaldi</LastName>
<Country>Italy</Country>
<Link>Vivaldi.xml</Link>
</Composer>
<Composer>
<FirstName>Frederick</FirstName>
<LastName>Delius</LastName>
<Country>England</Country>
<Link>Delius.xml</Link>
</Composer>
<Composer>
<FirstName>Johann Sebastian</FirstName>
```

Listing 7.6 A Catalog of Composers with Additional Information (Composers02.xml).

```
<LastName>Bach</LastName>
<Country>Germany</Country>
<Link>BachJS.xml</Link>
</Composer>
</Composers>
```

Listing 7.6 (*Continued*)

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output
  method="xml"
  indent="yes"
  />
<xsl:strip-space elements="*"/>
<xsl:template match="/">
<xsl:element name="Composers">
<xsl:call-template name="SortByCountry"/>
</xsl:element>
</xsl:template>

<xsl:template name="SortByCountry">
<xsl:for-each select="Composers/Composer">
<xsl:sort select="Country"/>
<xsl:sort select="LastName"/>
<xsl:copy-of select="."/>
</xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Listing 7.7 A Stylesheet to Sort Composers by Country and Last Name (SortComposers.xsl).

Thus we have a sorted input file, Listing 7.8, to use as input for a further transformation. To that source document we apply Listing 7.9.

This works by creating the skeleton for an unordered list. Nested within that, all Country element nodes are sorted using the value of the <Country> element as the primary key for sorting and the value of the LastName element node (which is a child of the parent node of the Country element node) as the secondary sort key, as shown in the <xsl:apply-templates> element from the main template.

```
<?xml version="1.0" encoding="utf-8"?>
<Composers>
   <Composer>
      <FirstName>Wolfgang</FirstName>
      <LastName>Mozart</LastName>
      <Country>Austria</Country>
      <Link>Mozart.xml</Link>
   </Composer>
   <Composer>
      <FirstName>Antonin</FirstName>
      <LastName>Dvorak</LastName>
      <Country>Czechoslovakia</Country>
      <Link>Dvorak.xml</Link>
   </Composer>
   <Composer>
      <FirstName>Frederick</FirstName>
      <LastName>Delius</LastName>
      <Country>England</Country>
      <Link>Delius.xml</Link>
   </Composer>
   <Composer>
      <FirstName>Johann Sebastian</FirstName>
      <LastName>Bach</LastName>
      <Country>Germany</Country>
      <Link>BachJS.xml</Link>
   </Composer>
   <Composer>
      <FirstName>Ludwig</FirstName>
      <LastName>Beethoven</LastName>
      <Country>Germany</Country>
      <Link>Beethoven.xml</Link>
   </Composer>
   <Composer>
      <FirstName>Antonio</FirstName>
      <LastName>Vivaldi</LastName>
      <Country>Italy</Country>
      <Link>Vivaldi.xml</Link>
   </Composer>
   <Composer>
      <FirstName>Peter</FirstName>
      <LastName>Tchaikovsky</LastName>
      <Country>Russia</Country>
      <Link>Tchaikovsky.xml</Link>
   </Composer>
</Composers>
```

TEAMFLY

Listing 7.8 A Sorted Catalog of Composers Expressed in XML (ComposersSorted.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"/>
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title><xsl:value-of select="name(/*)"/></title>
</head>
<body>
<p>On this site you can access information, by nationality, about some
  of the world's most famous composers.</p>
<ul>
<xsl:apply-templates select="/Composers/Composer/Country">
<xsl:sort select="."/>
<xsl:sort select="../LastName"/>
</xsl:apply-templates>
</ul>
</body>
</html>
</xsl:template>

<xsl:template match="Country">
<xsl:if test="not(.=preceding::Country)">
<li><xsl:value-of select="."/>:</li>
</xsl:if>
<ul>
<li><a href="{../Link}"> <xsl:value-of select="../LastName"/>,
<xsl:value-of select="../FirstName"/></a><br /></li>
</ul>
</xsl:template>

</xsl:stylesheet>
```

Listing 7.9 A Stylesheet to Create an HTML Document Listing Composers by Country (Composers02.xsl).

```
<xsl:apply-templates select="/Composers/Composer/Country">
<xsl:sort select="."/>
<xsl:sort select="../LastName"/>
</xsl:apply-templates>
```

The template that matches on Country uses an <xsl:if> element to test whether the preceding Country element node is different from the present Country element node. If it is different, then an <li> element is inserted in the outer <ul> element. If the two Country element nodes are the same, an <li> element is not created.

```
<xsl:template match="Country">
<xsl:if test="not(.=preceding::Country)">
<li><xsl:value-of select="."/>:</li>
</xsl:if>
<ul>
<li><a href="{../Link}"> <xsl:value-of select="../LastName"/>,
<xsl:value-of select="../FirstName"/></a><br /></li>
</ul>
</xsl:template>
```

Next, a nested unordered list is created. The first list item contains an <a> element whose href attribute is created using an attribute value template, whose value is the content of the Link element node, which is a child of the node, which is the parent node of the context node.

Within the <a> element, the last name of the composer is nested, separated by a comma from the composer's first name.

The output of the transformation is shown in Figure 7.2.

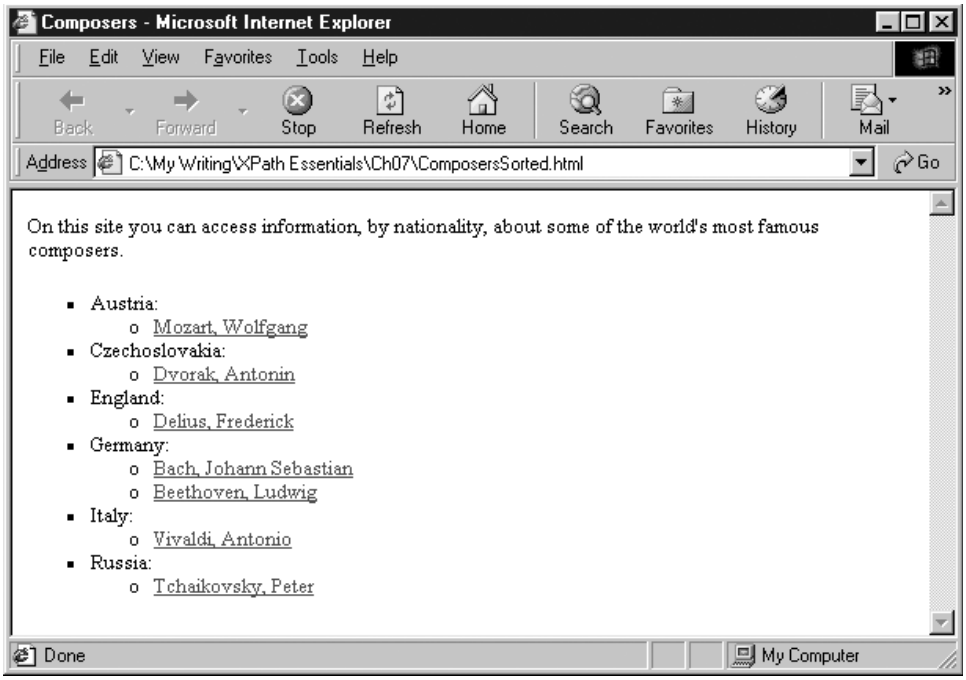


Figure 7.2 Creating Nested Lists in HTML.

Creating an HTML Table

Let's look at how to create an HTML table that outputs selected elements and attributes on the basis of a few XPath expressions. Our source document will be a brief summary of semi-finalists at the July 2001 Wimbledon tennis tournament, as shown in Listing 7.10.

To create a table to display the element content and the value of each country attribute, we can use the XSLT stylesheet in Listing 7.11.

```
<?xml version='1.0'?>
<SemiFinalists tournament="Wimbledon" date="July 2001">
<Player country="Croatia">Goran Ivanisovic</Player>
<Player country="UK">Tim Henman</Player>
<Player country="Australia">Pat Rafter</Player>
<Player country="USA">Andre Agassi</Player>
</SemiFinalists>
```

Listing 7.10 Wimbledon Men's Singles Semifinalists 2001 (SemiFinalists.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title><xsl:value-of select="SemiFinalists/@tournament"/> semi-finalists
  <xsl:value-of select="SemiFinalists/@date"/></title>
</head>
<body>
<br />
<table cellpadding="5" border="3">
<tr><td>Player</td>
<td>Country</td>
</tr>
<xsl:apply-templates select="SemiFinalists/Player"/>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="Player">
```

Listing 7.11 A Stylesheet to Create an HTML Table (SemiFinalists.xsl).

(*continues*)

```
<tr>
<td><xsl:value-of select="."/></td>
<td><xsl:value-of select="@country"/></td>
</tr>
</xsl:template>
</xsl:stylesheet>
```

Listing 7.11 (*Continued*)

The content of the <title> element is created using two <xsl:value-of> elements, which use abbreviated relative syntax to output the value of the tournament attribute of the <SemiFinalists> element, followed by the date attribute of the same element.

```
<title><xsl:value-of select="SemiFinalists/@tournament"/> semi-finalists
<xsl:value-of select="SemiFinalists/@date"/></title>
```

Within the body of the HTML page, we create a table using a <table> element in the normal way and also create a header row within the main template. I used <tr> tags, although you could use <th> tags, if you prefer.

In many cases we won't know how many elements exist or how many will be output in the table. Thus we need a mechanism to create a row for each such element. Of course, in a semifinals situation it is obvious that we need four rows, but the design is not dependent on knowing that.

In our example, in each row of the table we want to output the name of the player and his country of origin. We can do that by using <xsl:apply-templates> to instantiate a template to create a table row using a <tr> element and its accompanying <td> elements. In the first table cell we can use the simplest XPath location path, the full stop representing the context node, to output the content of a <Player> element. Similarly we use the `@country` location path to output the value of the country attribute in the second cell of each row.

Once each of the templates applied by the <xsl:apply-templates> element has been processed, then control returns to the main template, where we add the </table> tag, thus closing our table. If we had wanted to create a table footer then we could have added that immediately before the </table> tag.

In the above example we hard coded the number of columns in the table. For simple examples, that is perfectly adequate. But suppose we wanted to create a table that would allow us to display sales figures for a variable number of periods and adjust the number of columns in the resulting HTML table according to the number of pieces of data we had available to display?

Our source XML document is shown in Listing 7.12 and describes sales figures by office by region for XMML.com.

The output from the XSLT stylesheet is shown in Figure 7.3.

The code that produced that output from the XML source document is shown in Listing 7.13.

```
<?xml version='1.0'?>
<XMML:Sales xmlns:XMML="http://www.XMML.com/Finance">
<Location>New York
<SalesQ12002>135</SalesQ12002>
<SalesQ22002>141</SalesQ22002>
<SalesQ32002>150</SalesQ32002>
</Location>
<Location>London
<SalesQ12002>37</SalesQ12002>
<SalesQ22002>42</SalesQ22002>
<SalesQ32002>35</SalesQ32002>
</Location>
<Location>Tokyo
<SalesQ12002>220</SalesQ12002>
<SalesQ22002>181</SalesQ22002>
<SalesQ32002>230</SalesQ32002>
</Location>
<Location>Berlin
<SalesQ12002>101</SalesQ12002>
<SalesQ22002>99</SalesQ22002>
<SalesQ32002>113</SalesQ32002>
</Location>
</XMML:Sales>
```

Listing 7.12 Sales Figures by Location and Date (XMMLSales.xml).

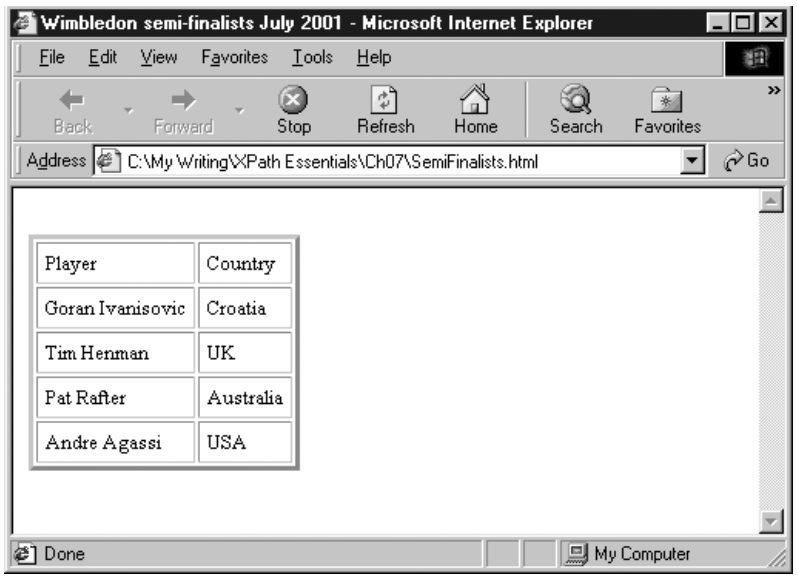


Figure 7.3 Creating a Table Using XSLT Variables.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:XMML="http://www.XMML.com/Finance"
                >
<xsl:output method="html"
            indent="yes"
            />
<xsl:strip-space elements="*"/>
<xsl:variable name="Columns"
select="count(XMML:Sales/Location[position()=1]/child::*)"/>
<xsl:template match="/">
<html>
<head>
<title>Sales Figures by Location</title>
                                  </head>
<body>
<h3>Sales for XMML.com from <xsl:value-of
select="substring(name(XMML:Sales/Location[1]/*[1]), 6, 2)"/>
<xsl:text> </xsl:text>
<xsl:value-of select="substring(name(XMML:Sales/Location[1]/*[1]), 8,
  4)"/> to
<xsl:value-of select="substring(name(XMML:Sales/Location[1]/*[last()]),
  6, 2)"/><xsl:text> </xsl:text>
<xsl:value-of select="substring(name(XMML:Sales/Location[1]/*[last()]),
  8, 4)"/></h3>
<br />
<table cellspacing="1" border="1px" width="100%">
<tr><td width="25%" align="left"><b>Location</b></td>
<xsl:for-each select="XMML:Sales/Location[1]/child::*">
<td align="left" width="{round(75 div $Columns)}%" >
<b><xsl:value-of select="substring(name(.), 6 , 2)"/>:<xsl:value-of
  select="substring(name(.), 8 , 4)"/></b>
</td>
</xsl:for-each>
</tr>
<xsl:apply-templates select="XMML:Sales/Location"/>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="Location">
<tr>
<td align="left"><xsl:value-of select="text()"/></td>
<xsl:for-each select="child::*">
```

Listing 7.13 A Stylesheet to Create HTML Table Using XSLT Variables (XMMLSales.xsl).

```
<td><xsl:value-of select="."/></td>

</xsl:for-each>
</tr>
</xsl:template>

</xsl:stylesheet>
```

Listing 7.13 (*Continued*)

In order to determine how many columns should be built into the table, we need to count the number of child elements contained in the <Location> element. We can do that using the count ( ) function. We need to select only the element children, remembering that the Location element node also has a text node child. We can do so using the location step `child::*` or `*` for short, remembering that the principal node type for the child axis is the element node.

```
<xsl:variable name="Columns" select="count(XMML:Sales/Location
  [position()=1]/child::*)"/>
```

Thus, with the source document having three child elements for each <Location> element the XSLT variable, called $Columns, will have a value of 3. We will use that value later in the stylesheet.

In creating the heading for the Web page, we make extensive use of the substring ( ) function to slice up the names of the element children of the <Location> element.

```
<h3>Sales for XMML.com from <xsl:value-of
select="substring(name(XMML:Sales/Location[1]/*[1]), 6, 2)"/>
<xsl:text> </xsl:text>
<xsl:value-of select="substring(name(XMML:Sales/Location[1]/*[1]), 8,
  4)"/> to
<xsl:value-of select="substring(name(XMML:Sales/Location[1]/*[last()]),
  6, 2)"/><xsl:text> </xsl:text>
<xsl:value-of select="substring(name(XMML:Sales/Location[1]/*[last()]),
  8, 4)"/></h3>
```

We could have used attributes to hold the same information, but the structure within the element type names gives us a good excuse to try out the substring ( ) function. Notice that the substring ( ) function is being applied to the result of the name ( ) function, which is nested within it.

The first substring ( ) function uses part of the element type name of the first element child of the first <Location> element and extracts the two characters that follow the sixth character. With our source code those two characters would be “Q1” as you saw in Figure 7.3. The second substring ( ) function uses the four characters that follow the eighth character. With our source document, those four characters would be “2002”.

Similarly, the third and fourth substring ( ) functions extract the corresponding information from the last child element of the first <Location> element. Assuming that the child elements are ordered by date, then this gives us the end of the period for which a report is being constructed in HTML.

Let's move on and look at how the HTML table is constructed. The following code begins the process:

```
<table cellspacing="1" border="1px" width="100%">
<tr><td width="25%" align="left"><b>Location</b></td>
<xsl:for-each select="XMML:Sales/Location[1]/child::*">
<td align="left" width="{round(75 div $Columns)}%" >
<b><xsl:value-of select="substring(name(.), 6 , 2)"/>:<xsl:value-of
  select="substring(name(.), 8 , 4)"/></b>
</td>
</xsl:for-each>
</tr>
```

The first two lines are straightforward HTML. The <xsl:for-each> element creates a table cell for each of the child element nodes of the Location element node. To calculate the column width for each column, we divide 75 percent (100 percent less the width of the "Location" column) by the value of the $Columns XSLT variable. In our example, since there are three child elements of the <Location> element, we have three columns created, each of 25 percent. We again use the substring ( ) function to extract the text "Q1" and "2002" and separate them by a colon. Similar text is extracted for each of the remaining columns created in the table.

The remaining rows of the table are constructed using a template that matches on Location:

```
<xsl:template match="Location">
<tr>
<td align="left"><xsl:value-of select="text()"/></td>
<xsl:for-each select="child::*">
<td><xsl:value-of select="."/></td>

</xsl:for-each>
</tr>
</xsl:template>
```

For the leftmost of the four columns in each row, we use the value of the text node child of the Location element node. For the content of each of the other three cells in each row of the table, we simply use the value of the content of each child element.

Control returns to the final part of the main template, and our HTML table is closed tidily.

Try creating additional elements for the report period or altering the quarters or years on which reports are made, but remember to keep the elements in date order and use the same structure to the element type names. Otherwise, the substring ( ) functions won't work as intended.

Creating a Pseudo Schema in HTML

Let's look at creating a listing of all the elements present in an XML source document. We won't attempt to create a full schema, but simply provide a tabulated listing in HTML. This gives us an opportunity to use a couple of the namespace-related functions.

First let's create a source document, shown in Listing 7.14, with lots of namespace goodies in it.

As you can see, there are three different namespaces used, although only two are on elements.

Listing 7.15 shows the XSLT stylesheet that we will use to create the output HTML table.

```
<?xml version='1.0'?>
<XMML:Document xmlns:XMML="http://www.XMML.com/Documentation/">
<XMML:Introduction>The document begins here.</XMML:Introduction>
<XMML:MainText>The main ideas are expressed here.</XMML:MainText>
<XMML:Illustration xlink:href="Rectangle.svg"
xmlns:xlink="http://www.w3.org/1999/xlink">
<svg:svg width="100%" height="100%"
xmlns:svg="http://www.w3.org/2000/svg">
<svg:rect x="50" y="75" width="100px" height="100px" style="stroke:red;
  fill:none;"/>
</svg:svg>
</XMML:Illustration>
<XMML:ExternalElement>
<SharePrice>$100</SharePrice>
</XMML:ExternalElement>
<XMML:Note>Remember to communicate ideas clearly.</XMML:Note>
<XMML:Epilog>I said what I intended to say.</XMML:Epilog>
</XMML:Document>
```

Listing 7.14 A Representation in XML of a Document That Uses Namespaces (XMML-Document.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
```

Listing 7.15 A Stylesheet to Create an HTML Table of the XML Elements in the Source Document (PseudoSchema.xsl). (*continues*)

```
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using namespace-related functions</title>
</head>
<body>
<h3>A tabulated list of all elements in the source document.</h3>
<table width="100%" border="1" cellspacing="2">
<tr>
<td width="25%">Element Type Name</td>
<td width="15%">Namespace Prefix</td>
<td width="25%">Local Part</td>
<td width="35%">Namespace URI</td>
</tr>
<xsl:apply-templates select="//*">
<xsl:sort select="namespace-uri()"/>
<xsl:sort select="local-name()"/>
</xsl:apply-templates>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="*">
<xsl:variable name="NSPrefix">
<xsl:if test="contains(name(.), ':')">
<xsl:value-of select="substring-before(name(.), ':')"/>
</xsl:if>
</xsl:variable>
<tr>
<td><xsl:value-of select="name(.)"/></td>
<xsl:choose>
<xsl:when test="not($NSPrefix='')">
<td><xsl:value-of select="$NSPrefix"/></td>
</xsl:when>
<xsl:otherwise>
<td> </td>
</xsl:otherwise>
</xsl:choose>
<td><xsl:value-of select="local-name(.)"/></td>
<xsl:choose>
<xsl:when test="not($NSPrefix='')">
<td><xsl:value-of select="namespace-uri(.)"/></td>
</xsl:when>
<xsl:otherwise>
<td> </td>
```

TEAMFLY

Listing 7.15 (*Continued*)

```
</xsl:otherwise>
</xsl:choose>
</tr>
</xsl:template>
</xsl:stylesheet>
```

Listing 7.15 (*Continued*)

Within the main template, the HTML table is created and its first row is populated with labels. Then the <xsl:apply-templates> element selects all element nodes that are descendants of the root node (that is, all element nodes that are in the document).

```
<xsl:apply-templates select="//*">
<xsl:sort select="namespace-uri()"/>
<xsl:sort select="local-name()"/>
</xsl:apply-templates>
```

Those element nodes are sorted with their namespace URI as returned by the namespace-uri ( ) function as the primary sort key and the local part as returned by the local-name ( ) function as the secondary sort key.

NOTE Notice the potentially confusing difference in terminology between the XML Namespaces terminology, which refers to the part of an element type name after the colon as the "local part," while the local-name ( ) function is the term used for the corresponding XPath function.

The output HTML document is shown in Listing 7.16. If you don't already routinely use the <xsl:output> element with the indent attribute set to "yes", the neat output from Instant Saxon of this fairly lengthy HTML document should persuade you of the advantages. I suggest at the same time that you use the <xsl:strip-space> element as well.

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8">

      <title>Using namespace-related functions</title>
   </head>
   <body>
      <h3>A tabulated list of all elements in the source document.</h3>
      <table width="100%" border="1" cellspacing="2">
         <tr>
```

Listing 7.16 The HTML Document Created by the Stylesheet in Listing 7.15 (Pseudo-Schema.html). (*continues*)

```
            <td width="25%">Element Type Name</td>
            <td width="15%">Namespace Prefix</td>
            <td width="25%">Local Part</td>
            <td width="35%">Namespace URI</td>
         </tr>
         <tr>
            <td>SharePrice</td>
            <td> </td>
            <td>SharePrice</td>
            <td> </td>
         </tr>
         <tr>
            <td>svg:rect</td>
            <td>svg</td>
            <td>rect</td>
            <td>http://www.w3.org/2000/svg</td>
         </tr>
         <tr>
            <td>svg:svg</td>
            <td>svg</td>
            <td>svg</td>
            <td>http://www.w3.org/2000/svg</td>
         </tr>
         <tr>
            <td>XMML:Document</td>
            <td>XMML</td>
            <td>Document</td>
            <td>http://www.XMML.com/Documentation/</td>
         </tr>
         <tr>
            <td>XMML:Epilog</td>
            <td>XMML</td>
            <td>Epilog</td>
            <td>http://www.XMML.com/Documentation/</td>
         </tr>
         <tr>
            <td>XMML:ExternalElement</td>
            <td>XMML</td>
            <td>ExternalElement</td>
            <td>http://www.XMML.com/Documentation/</td>
         </tr>
         <tr>
            <td>XMML:Illustration</td>
            <td>XMML</td>
            <td>Illustration</td>
            <td>http://www.XMML.com/Documentation/</td>
```

Listing 7.16 (*Continued*)

```
         </tr>
         <tr>
            <td>XMML:Introduction</td>
            <td>XMML</td>
            <td>Introduction</td>
            <td>http://www.XMML.com/Documentation/</td>
         </tr>
         <tr>
            <td>XMML:MainText</td>
            <td>XMML</td>
            <td>MainText</td>
            <td>http://www.XMML.com/Documentation/</td>
         </tr>
         <tr>
            <td>XMML:Note</td>
            <td>XMML</td>
            <td>Note</td>
            <td>http://www.XMML.com/Documentation/</td>
         </tr>
      </table>
   </body>
</html>
```

Listing 7.16 (*Continued*)

Did you notice that there is nothing within the stylesheet, PseudoSchema.xsl, that ties it to any particular source XML document? It would be possible to apply it to any XML source document, including itself. You might want to try that out.

If you are creating HTML using XPath and XSLT more than very occasionally, I suggest you think about having a stylesheet template that you can work from, in order to save your having to type the same elements time after time. The type of template that would precisely suit your needs is up to you to decide, but you might find Listing 7.17 helpful.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
```

Listing 7.17 A Skeleton Stylesheet for the Creation of HTML Output (XSLTToHTMLSkeleton.xsl). (*continues*)

```
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title></title>
</head>
<body>

</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

Listing 7.17 (*Continued*)

Looking Ahead

The examples in this chapter have shown you the use of several XPath functions in the creation of HTML. Together with the techniques to enable you to select elements and attributes, which you have seen in passing in this chapter and in many earlier examples, XPath gives you great flexibility to create HTML documents using XSLT.

Having looked at the conversion of XML to HTML in this chapter, Chapter 8 will look at the creation of scalable vector graphics using XPath and XSLT.

Using XPath and XSLT to Produce SVG

This chapter will introduce you to the use of XPath with XSLT to create Scalable Vector Graphics (SVG), based on the W3C's graphics standard.

As the information that businesses have to process becomes greater in volume and more demanding in complexity, access to visual representations of data will help them better understand what is happening within data, whether it is their own data or someone else's. SVG is a graphics format that is designed to satisfy a number of design and programming needs. SVG is expressed in XML syntax and has facilities for display on screen and on paper, and has declarative animation and facilities for interactivity, using either its own declarative syntax or JavaScript to manipulate SVG objects.

The SVG specification can be accessed at www.w3.org/SVG/.

Introduction to SVG

Scalable Vector Graphics is the W3C's XML-based graphics specification designed to replace bitmap graphics on the Web and elsewhere. SVG is, as its name suggests, a vector graphics language.

Vector graphics languages describe a graphic's shape by a series of instructions, which indicate how a shape is to be drawn. A rendering engine, also known as an SVG viewer, translates those instructions into a specific visual appearance. The instructions may be reinterpreted at any time allowing zooming of the graphic while maintaining

visual quality. This contrasts with bitmap graphics where a shape is simply a small or large two-dimensional array of pixels of varying color values, which, if zoomed, will increasingly be seen as an array of colored squares.

Early vector graphics languages were limited to relatively crude renditions of shapes. SVG has the ability to describe and display a sophisticated range of images using 24-bit RGB color, as well as providing a basic range of graphics primitives such as rectangles and ellipses. SVG provides elements for the creation of linear and radial gradients as well as a useful and powerful range of filter primitives which, by adjusting the values of their attributes and combining filter primitives in various sequences, allow the generation of sophisticated and esthetically satisfying graphics images. SVG provides a full range of transparency, continuously variable from 0 to 100 percent on a full range of 24-bit colors, which contrasts favorably with the ability of GIF images to apply 100 percent transparency to a single, chosen color only.

SVG provides powerful animation facilities. Almost any facet of an SVG image or element within such an image can be animated using declarative animation elements from the Synchronized Multimedia Integration Language (SMIL) 2.0 specification. SMIL 2.0 is fully described at www.w3.org/TR/smil20 and the SMIL Animation specification is described at www.w3.org/TR/smil-animation. Among the aspects of an SVG element that can be animated are position, size, color, visibility, or transparency.

In addition to the very useful facilities for declarative animation, SVG can be scripted using, for example, JavaScript to manipulate the SVG Document Object Model. The use of scripting opens up new possibilities of providing sophisticated programmatic control for user-initiated interactivity within graphic images.

The SVG specification is substantially longer than this book; therefore, this chapter can only give you a glimpse of SVG's capabilities, and more specifically how XPath can be used to create some fairly straightforward SVG images. SVG provides powerful general graphic facilities for the creation of static and animated graphics, which can be based on data stored as XML. XPath allows the selection of appropriate data to be displayed.

The display of such business data will typically feature a variety of SVG-based charts. SVG charts can be produced by export from established commercial programs or can be produced from raw XML using XPath and XSLT.

We will use both bar and line charts in the examples we will create later in the chapter. Two of the most relevant SVG elements in the production of such charts are the <line> element and the <rect> element. In addition, we will need a basic understanding of the SVG <text> element.

The SVG <line> Element

The SVG <line> element describes a straight line. It has four attributes. The x1 and y1 attributes define the position of one end of the line. The x2 and y2 attributes define the position of the other end of the line.

In addition, the <line> element can have a style attribute, which has, as its value, a number of CSS properties such as stroke color, stroke width, etc. The information in the style attribute defines the color, thickness, opacity, etc., of the line. An alternative approach to styling is to use an external CSS stylesheet with rules that are associated with

all <line> elements or a subclass of them, by selectively adding a class attribute to a <line> element.

In common with many other SVG elements, the <line> element can be animated using the <animate>, <animateColor>, <set>, <animateTransform>, or <animateMotion> elements.

In Listing 8.1, we will create four lines, which make up a simple rectangle using the <line> element.

As you can see, an SVG document may start with an XML declaration and may include a DOCTYPE declaration. The DOCTYPE declaration used in Listing 8.1 is that for the SVG Proposed Recommendation of July 2001. Check the URL www.w3.org/TR/SVG to determine whether an update to that DOCTYPE declaration is available.

The document element for all SVG images, or documents, is the <svg> element. The <svg> element may optionally have width and height attributes, which may use a variety of units. In Listing 8.1 the value of the width attribute is expressed as a percentage of the width of the browser window, and the value of the height attribute is expressed in pixels.

Assuming that we start drawing at the top left of the rectangle created by the four <line> elements, the x1 and y1 attributes of the first <line> element describe the x and y coordinates, respectively, of the starting point of the first line. The x2 and y2 attributes of that line element describe the x and y coordinates of the other end of that line (the top right of the rectangle created). If you are accustomed to using normal geometric coordinates, then the value of the y coordinates may have come as a minor surprise. SVG counts the top left of the containing <svg> element as (0,0). As you move to the right, the value of the x coordinate (or corresponding attribute) increases and as you move down, the value of the y coordinate (or corresponding attribute) also increases. This convention is the same as that found in other screen-oriented languages such as JavaScript.

In Figure 8.1 you can see the rectangle created using the four <line> elements.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/PR-SVG-20010719/DTD/svg10.dtd">

<svg width="100%" height="550px" xmlns="http://www.w3.org/2000/svg">
<line x1="100" y1="100" x2="600" y2="100" style="stroke:red;
  fill:red;"/>
<line x1="600" y1="100" x2="600" y2="300" style="stroke:red;
  fill:red;"/>
<line x1="600" y1="300" x2="100" y2="300" style="stroke:red;
  fill:red;"/>
<line x1="100" y1="300" x2="100" y2="100" style="stroke:red;
  fill:red;"/>
</svg>
```

Listing 8.1 A Simple SVG Document Creating a Rectangle Using Four <line> Elements (FourLines.svg).

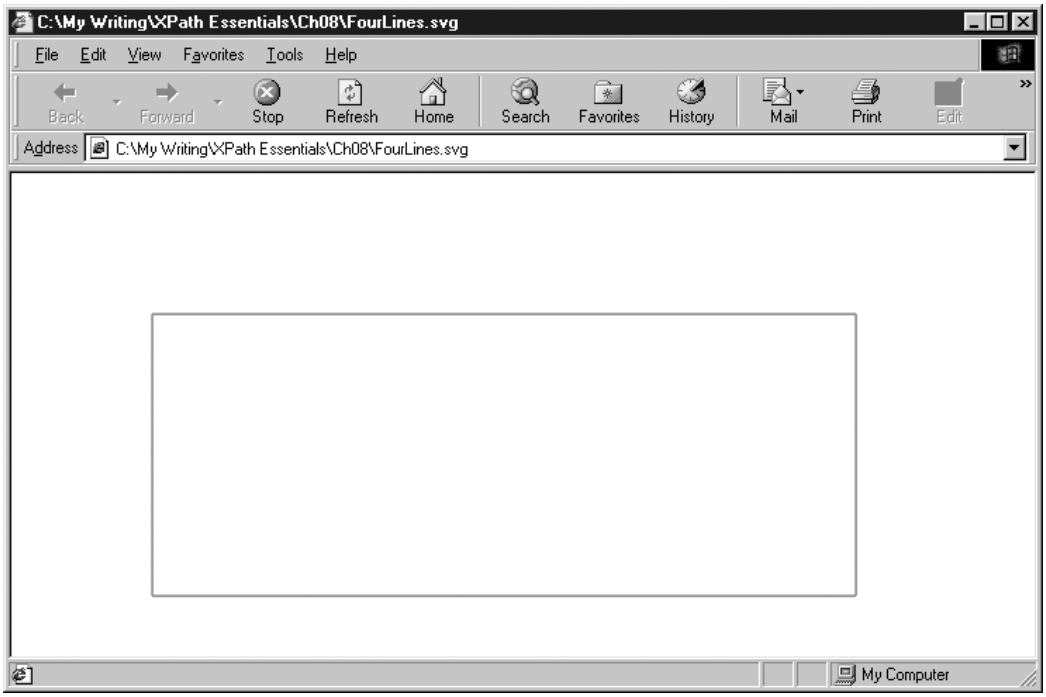

C:\My Writing\XPath Essentials\Ch08\FourLines.svg
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address C:\My Writing\XPath Essentials\Ch08\FourLines.svg
My Computer

Figure 8.1 The Rectangle Created Using Listing 8.1.

The SVG <rect> Element

The SVG <rect> element is used to describe the parameters of a rectangular shape. Since a square is simply a rectangle where the width and height are equal, the <rect> element is sufficient to describe all four-sided rectilinear objects.

The <rect> element possesses both x and y attributes. The x and y attributes describe the coordinates of the top left corner of the rectangle described by the <rect> element. The <rect> element has width and height attributes, which define the width and height of the rectangle. By default a rectangle has rectangular corners, but rounded corners are created by use of the rx and ry attributes of the <rect> element.

The styling of a <rect> element is achieved using a style attribute with a collection of CSS properties as the value of that attribute or by using an external CSS stylesheet. The properties in the style attribute describe the color, width, opacity, etc., of the stroke of the outline of the rectangle and may also describe the color, opacity, etc., of the rectangle's fill.

An alternative approach to creating the rectangle that we created in Listing 8.1 would be to use the SVG <rect> element. To create a rectangle with on-screen appearance identical to that shown in Figure 8.1, but this time using the <rect> element, we could use the code shown in Listing 8.2.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20001102//EN"
     "http://www.w3.org/TR/2000/CR-SVG-20001102/DTD/svg-20001102.dtd">

<svg width="100%" height="550px" xmlns="http://www.w3.org/2000/svg">
<rect x="100" y="100" width="498" height="198" style="stroke:red;
  stroke-width:2; fill:none"/>
</svg>
```

Listing 8.2 An SVG Document Using the <rect> Element (SimpleRect.svg).

We will use both the <line> and <rect> element in the later examples in this chapter.

The SVG <text> Element

In order to create labels and title text for the graphs we will create later in the chapter, we need to add some text to them. SVG provides three elements that can be used in relation to display of text: the <text>, <tspan>, and <tref> elements. For the purposes of our graphs, the <text> element has the functionality we need and will be the only text-related SVG element described here.

The <text> element has x and y attributes that define the *lower* left corner of the invisible box within which the text is displayed. In addition the style attribute of the <text> element contains a range of CSS properties, which define, for example, font size, font weight, etc.

If we wanted to display a simple message using the <text> element, we could use code like this:

```
<text x="50px" y="50px" style="font-family: Arial, sans-serif; font-
  size:14; font-weight:bold; fill:blue">Welcome to SVG!</text>
```

The visual appearance of a character is called a *glyph*. Characters are mapped to glyphs using character-to-glyph mapping tables, which are specific to each font. The above code displays the text using the Arial font or the default system sans-serif font. If we wished to use a serif font, such as Times New Roman, we could simply adjust the code as shown here:

```
<text x="50px" y="50px" style="font-family: 'Time New Roman', serif;
  font-size:14; font-weight:bold; fill:blue">Welcome to SVG!</text>
```

SVG Tools

To use SVG you need two types of tools. You need something to create SVG, and you need something with which to view SVG.

For our purposes we need only a text editor, ideally an XML-aware one, to create the SVG since we will simply be using XPath and XSLT to generate SVG documents or images from simple XML source documents.

The best SVG viewer available at the time of writing is the Adobe SVG Viewer. It can be downloaded, without charge, from www.adobe.com/svg/. Versions exist for Internet Explorer and Netscape 4.X on both the Windows and Macintosh platforms. At the time of writing, the Linux platform is not supported.

NOTE **the Adobe SVG Viewer on Netscape 6 and Opera 5 is not supported officially by Adobe. However, at least with version 2 of the Adobe viewer, if you copy three files—SVGViewer.zip, SVGView.dll, and NPSVGVw.dll—to the plugins directory for Netscape 6 or Opera 5, you should be able to view SVG images using those browsers.**

An alternative SVG viewer is the X-Smiles multi-namespace XML browser available for free download from www.xsmiles.org. As well as displaying SVG, the X-Smiles browser can display XSL-FO, SMIL, and XForms. However, at present, the SVG functionality in the Adobe Viewer is significantly more complete than in the X-Smiles browser.

If you want to view SVG, but do not use a platform supported by the Adobe SVG Viewer, a further option is the Batik SVG toolkit. It is a full Java-based SVG toolkit but does include a useful Java-based SVG viewer, which can be run without having to become involved in the detail of the programming toolkit. The Batik software can be downloaded without charge from http://xml.apache.org/.

TEAM-FLY

Creating a Static SVG Bar Chart

First, let's look at how we can create a bar chart describing sales figures for five (fictional) offices of XMML.com during the year 2001.

The annual sales figures for each office for 2001, expressed in U.S. dollars (000's), are shown in Listing 8.3.

The code in Listing 8.3 will be the source for the information that we will choose to display in our bar chart. Before we use XPath to select the data for display in the bar chart, let's look at how we would create the axes for the bar chart. A stylesheet that will create a skeleton SVG document to display simple vertical horizontal and vertical axes is shown in Listing 8.4.

```
<?xml version='1.0'?>
<OfficeSales year="2001">
<Office sales="250" abbreviation="NY">New York</Office>
<Office sales="750" abbreviation="SF">San Francisco</Office>
<Office sales="500" abbreviation="Ams">Amsterdam</Office>
<Office sales="1000" abbreviation="Lon">London</Office>
<Office sales="250" abbreviation="Par">Paris</Office>
</OfficeSales>
```

Listing 8.3 Sales Figures for 2001 by Office for XMML.com (OfficeSales.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/2000/svg">

<xsl:template match="/">
<svg width="100%" height="550px">
<rect x="0" y="0" width="100%" height="100%" style="fill:#DDDDDD"/>
<line x1="100" y1="50" x2="100" y2="500" style="stroke:black; fill:none;
  stroke-width:2;"/>
<line x1="100" y1="500" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:2;"/>

</svg>
</xsl:template>

</xsl:stylesheet>
```

Listing 8.4 A Stylesheet to Create an SVG Image with Vertical and Horizontal Axes Only (SVGSkeleton.xsl).

The stylesheet creates a simple SVG document with the code shown in Listing 8.5.

The SVG produced by the stylesheet consists of a pale gray background created using an SVG <rect> element plus the vertical and horizontal axes for the graph and two <line> elements. We create an SVG document using an XSLT stylesheet similarly to the way we have created HTML output in many examples in earlier chapters.

The visual appearance produced is shown in Figure 8.2.

We need to add axis marks and labels for the axes, as well as a label for the whole SVG chart. We will do this before applying XPath. To help you see what is happening, I have split the stylesheet up into a number of named templates, which are called using the <xsl:call-template> element. Each named template describes part of the creation of the SVG graph. The code for achieving that is shown in Listing 8.6.

```
<?xml version="1.0" encoding="utf-8"?>
<svg width="100%" height="550px" xmlns="http://www.w3.org/2000/svg">
<rect x="0" y="0" width="100%" height="100%" style="fill:#DDDDDD"/>
<line x1="100" y1="50" x2="100" y2="500" style="stroke:black; fill:none;
  stroke-width:2;"/>
<line x1="100" y1="500" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:2;"/>
</svg>
```

Listing 8.5 SVG Output of the Stylesheet in Listing 8.4 (SVGSkeleton.svg).

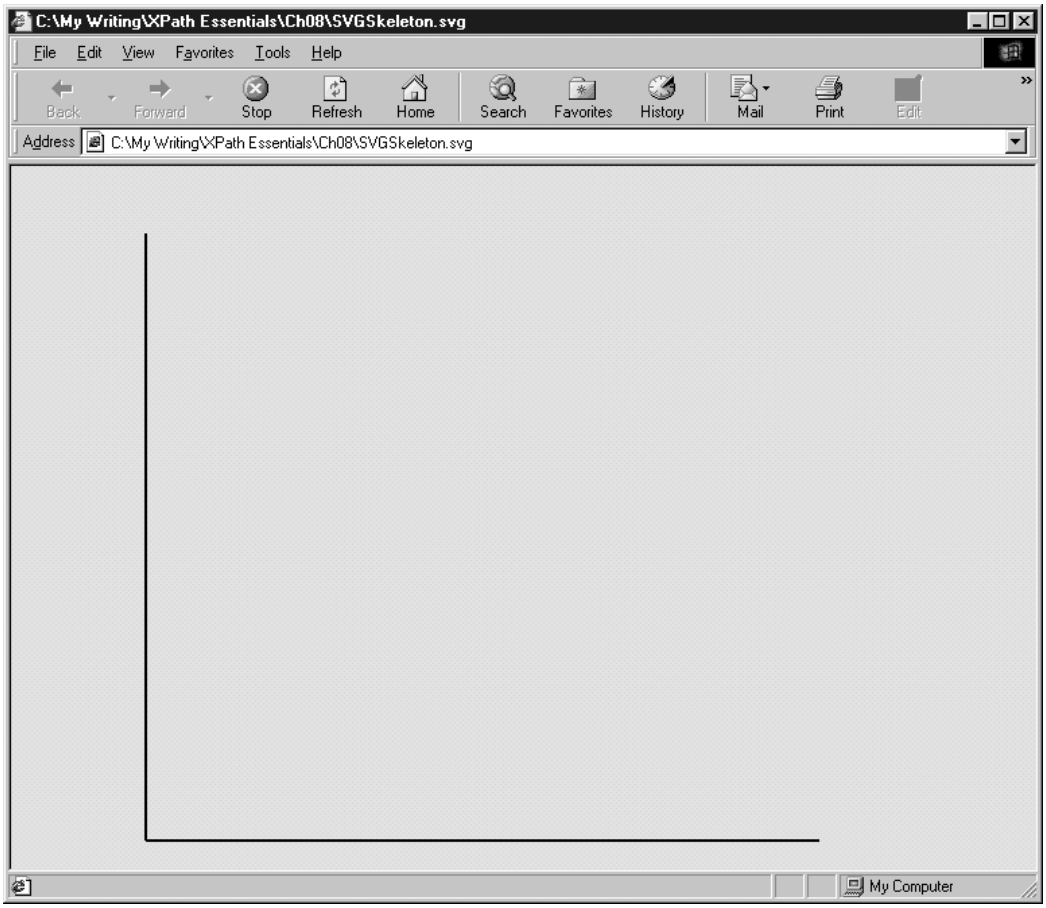

C:\My Writing\XPath Essentials\Ch08\SVGSkeleton.svg
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit
Address C:\My Writing\XPath Essentials\Ch08\SVGSkeleton.svg
My Computer

Figure 8.2 The Axes for the Graph Created Using the Code Shown in Listing 8.5.

If SVG is totally new to you, the amount of code might look intimidating, but if you follow the named templates it will tell you what is being created. Thereafter, the use of <line> and <text> elements should be self-explanatory.

The visual appearance produced by Listing 8.6 is shown in Figure 8.3.

Since the structure of our source XML document is straightforward, the use of XPath here is not complex.

We can associate the title of our graph to the year whose sales figures are being reported by using the <xsl:value-of> element, as here:

```
<text x="190" y="40" style="stroke:none; fill:#FF6600; font-size:18;
  font-weight:bold; font-family:Arial, sans-serif;">
XMML.com <xsl:value-of select="/OfficeSales/@year"/>Sales Figures by
  Office
</text>
```

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/2000/svg">

<xsl:template match="/">
<svg width="100%" height="550px">
<title>XMML.com Sales by Office</title>
<rect x="0" y="0" width="100%" height="100%" style="fill:#DDDDDD"/>
<xsl:call-template name="CreateAxes"/>

<xsl:call-template name="CreateGraphTitle"/>
</svg>
</xsl:template>

<xsl:template name="CreateAxes">
<xsl:call-template name="CreateVerticalAxis"/>
<xsl:call-template name="CreateHorizontalAxis"/>
</xsl:template>

<xsl:template name="CreateVerticalAxis">
<line x1="100" y1="50" x2="100" y2="500" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="90" y1="60" x2="100" y2="50" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="110" y1="60" x2="100" y2="50" style="stroke:black; fill:none;
  stroke-width:1;"/>
<!-- Create the axis marks for each office -->
<line x1="100" y1="75" x2="95" y2="75" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="175" x2="95" y2="175" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="275" x2="95" y2="275" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="375" x2="95" y2="375" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="475" x2="95" y2="475" style="stroke:black; fill:none;
  stroke-width:1;"/>
<!-- Create the labels for the Y axis [LATER] -->
</xsl:template>

<xsl:template name="CreateHorizontalAxis">
<line x1="100" y1="500" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="590" y1="490" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:1;"/>
```

Listing 8.6 Stylesheet Skeleton to Lay Out the Axes with Axis Marks and Labels (SVGSkeleton02.xsl). (continues)

```
<line x1="590" y1="510" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:1;"/>
<!-- Create the axis marks for the sales value. -->
<line x1="200" y1="500" x2="200" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="300" y1="500" x2="300" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="400" y1="500" x2="400" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="500" y1="500" x2="500" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<!-- Create the labels for the horizontal axis. -->
<text x="190" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
250
</text>
<text x="290" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
500
</text>
<text x="390" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
750
</text>
<text x="485" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
1000
</text>
</xsl:template>

<xsl:template name="CreateGraphTitle">
<!-- Add the title for the graph -->
<text x="190" y="40" style="stroke:none; fill:#FF6600; font-size:18;
  font-weight:bold; font-family:Arial, sans-serif;">
XMML.com Sales Figures by Office
</text>
</xsl:template>
</xsl:stylesheet>
```

Listing 8.6 (*Continued*)

We can create a text label for each axis mark on the vertical axis with code like this:

```
<text x="70" y="80" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=1]/@abbreviation"/>
</text>
```

We could use the value of the sales attribute in the source XML document to

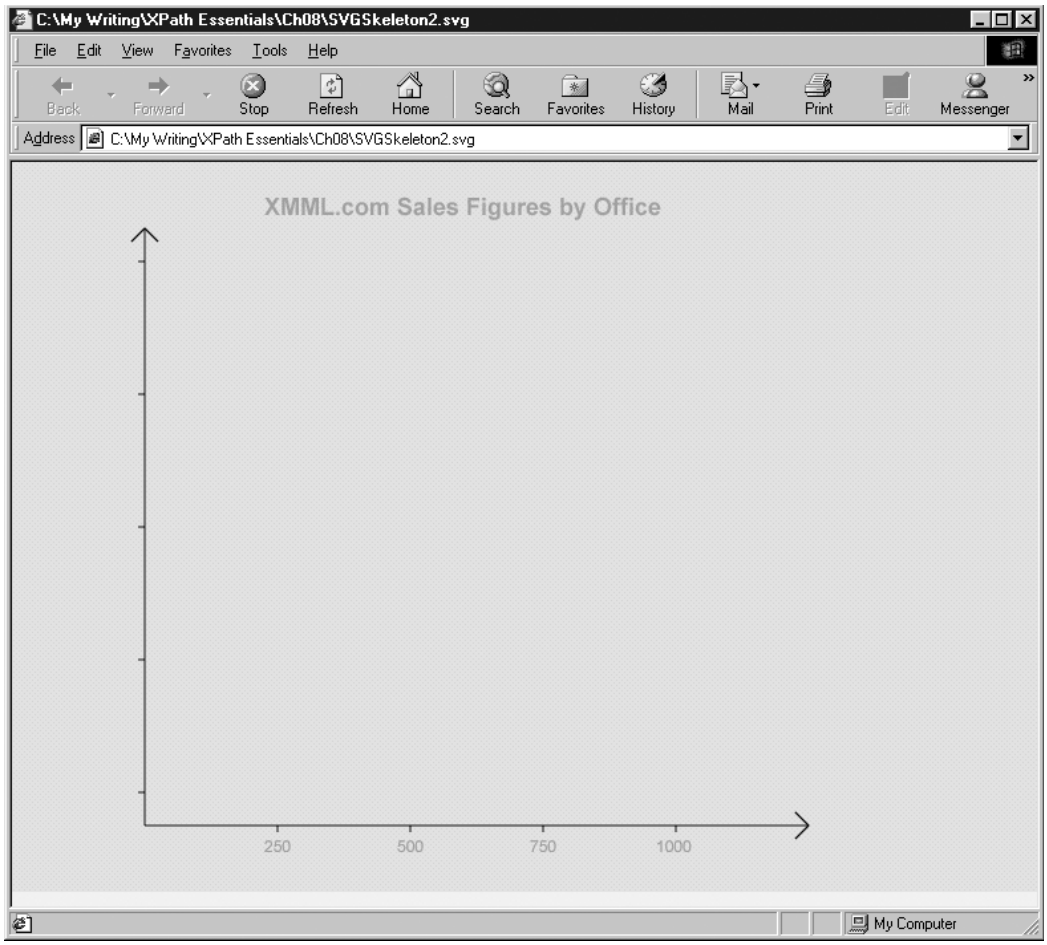


Figure 8.3 The SVG Axes, Axis Marks, and Labels.

calculate the width of the bar for each office. In the stylesheet, I have shown two ways to do this. The first uses XSLT variables with an attribute value template, as in the following code snippet:

```
<xsl:variable name="Bar1" select="/OfficeSales/Office[position()=1]/
  @sales"/>
<rect x="101" y="65" width="{0.4 * $Bar1}" height="20"
style="stroke:none; fill:#FF6600;"/>
```

Alternatively, an attribute value template could calculate the width of the bar directly, as here:

```
<rect x="101" y="165" width="{0.4 *
(/OfficeSales/Office[position()=2]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;"/>
```

The reason for applying a factor of 0.4 in each of the above code snippets is that a sales value of 250 (thousands of U.S. dollars) has to be mapped to a width of 100 pixels.

You have probably noticed that I have semi-hardcoded the position and dimension of the bars in the bar chart. The positions and dimensions of the bars, as well as the positions of the axis marks and labels for each axis, could be calculated using appropriate XSLT variables. This could allow us, for example, to add another office to the company or, in a more generic situation, to display quarterly sales figures or annual sales figures by month. However, for the current purposes, hard coding of part of the SVG image helps you see what is happening.

Listing 8.7 is the full code to create the finished bar chart.

In Figure 8.4 you can see the finished appearance of the bar chart.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/2000/svg">

<xsl:template match="/">
<svg width="100%" height="550px">
<title>XMML.com Sales by Office</title>
<rect x="0" y="0" width="100%" height="100%" style="fill:#DDDDDD"/>
<xsl:call-template name="CreateAxes"/>
<xsl:call-template name="CreateBars"/>
<xsl:call-template name="CreateGraphTitle"/>
</svg>
</xsl:template>

<xsl:template name="CreateAxes">
<xsl:call-template name="CreateVerticalAxis"/>
<xsl:call-template name="CreateHorizontalAxis"/>
</xsl:template>

<xsl:template name="CreateVerticalAxis">
<line x1="100" y1="50" x2="100" y2="500" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="90" y1="60" x2="100" y2="50" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="110" y1="60" x2="100" y2="50" style="stroke:black; fill:none;
  stroke-width:1;"/>
<!-- Create the axis marks for each office -->
<line x1="100" y1="75" x2="95" y2="75" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="175" x2="95" y2="175" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="275" x2="95" y2="275" style="stroke:black; fill:none;
  stroke-width:1;"/>
<line x1="100" y1="375" x2="95" y2="375" style="stroke:black; fill:none;
```

Listing 8.7 A Stylesheet to Create the Finished Horizontal Bar Chart (SVGChart.xsl).

```
  stroke-width:1;"/>
<line x1="100" y1="475" x2="95" y2="475" style="stroke:black; fill:none;
  stroke-width:1;"/>
<!-- Create the labels for the Y axis -->
<xsl:call-template name="CreateVerticalLabels"/>
</xsl:template>

<xsl:template name="CreateVerticalLabels">
<text x="70" y="80" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=1]/@abbreviation"/>
</text>
<text x="70" y="180" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=2]/@abbreviation"/>
</text>
<text x="70" y="280" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=3]/@abbreviation"/>
</text>
<text x="70" y="380" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=4]/@abbreviation"/>
</text>
<text x="70" y="480" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=5]/@abbreviation"/>
</text>
</xsl:template>

<xsl:template name="CreateHorizontalAxis">
<line x1="100" y1="500" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="590" y1="490" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="590" y1="510" x2="600" y2="500" style="stroke:black;
  fill:none; stroke-width:1;"/>
<!-- Create the axis marks for the sales value. -->
<line x1="200" y1="500" x2="200" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="300" y1="500" x2="300" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="400" y1="500" x2="400" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<line x1="500" y1="500" x2="500" y2="505" style="stroke:black;
  fill:none; stroke-width:1;"/>
<!-- Create the labels for the horizontal axis. -->
```

Listing 8.7 (*Continued*) (*continues*)

```
<text x="190" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
250
</text>
<text x="290" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
500
</text>
<text x="390" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
750
</text>
<text x="485" y="520" style="stroke:none; fill:#FF6600; font-size:12;
  font-family:Arial, sans-serif;">
1000
</text>
</xsl:template>

<xsl:template name="CreateBars">
<xsl:variable name="Bar1"
select="/OfficeSales/Office[position()=1]/@sales"/>
<rect x="101" y="65" width="{0.4 * $Bar1}" height="20"
style="stroke:none; fill:#FF6600;"/>
<rect x="101" y="165" width="{0.4 *
  (/OfficeSales/Office[position()=2]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;"/>
<rect x="101" y="265" width="{0.4 *
  (/OfficeSales/Office[position()=3]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;"/>
<rect x="101" y="365" width="{0.4 *
  (/OfficeSales/Office[position()=4]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;"/>
<rect x="101" y="465" width="{0.4 *
  (/OfficeSales/Office[position()=5]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;"/>
</xsl:template>

<xsl:template name="CreateGraphTitle">
<!-- Add the title for the graph -->
<text x="190" y="40" style="stroke:none; fill:#FF6600; font-size:18;
  font-weight:bold; font-family:Arial, sans-serif;">
XMML.com <xsl:value-of select="/OfficeSales/@year"/> Sales Figures by
  Office
</text>
</xsl:template>

</xsl:stylesheet>
```

Listing 8.7 (*Continued*)

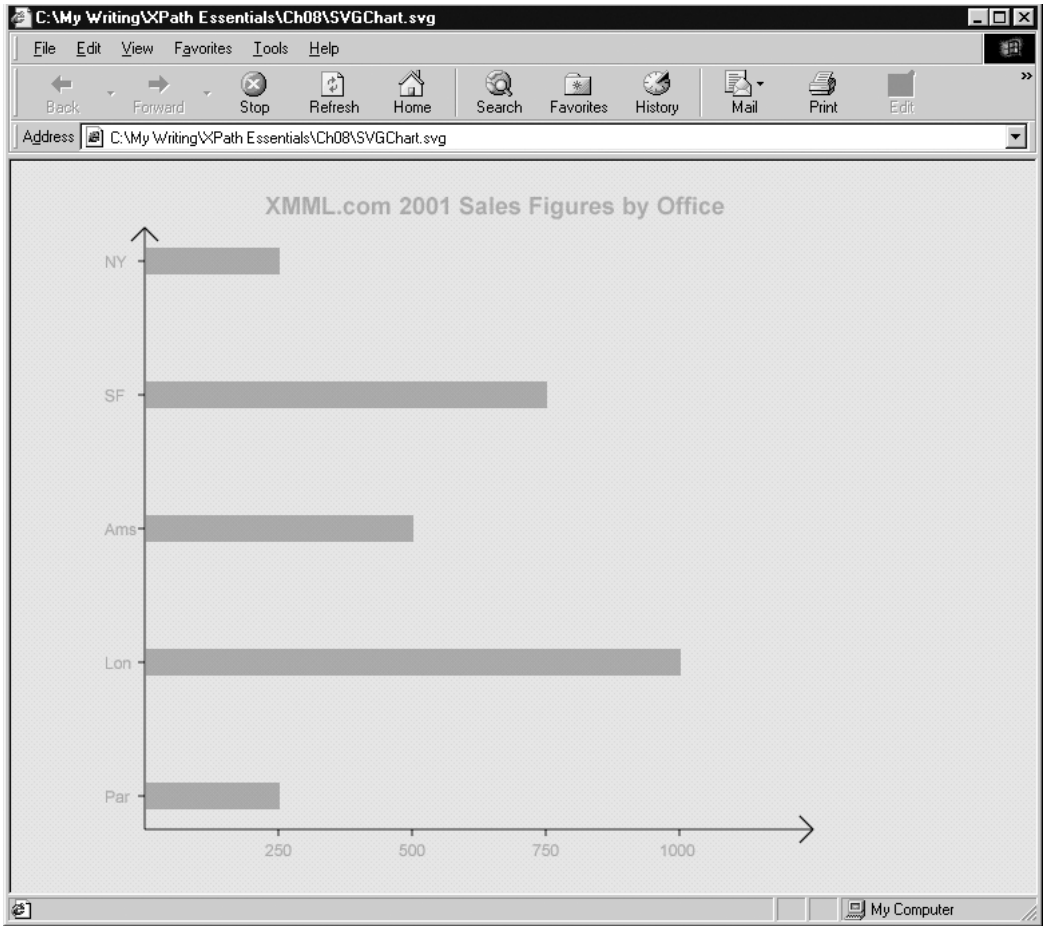


Figure 8.4 The Finished Bar Chart Showing Sales Figures for XMML.com.

Creating an Animated SVG Bar Chart

Adding animation to SVG elements is pretty straightforward, since most SVG elements can be animated using the SMIL Animation declarative animation elements.

The only part of the XSLT stylesheet that we need to change in order to create the animations in the following example is the CreateBars template, shown in Listing 8.8. As rewritten, it has five different SVG animations, which we will examine individually.

Here is the code for the animation of the bar representing sales at the New York office:

```
<xsl:variable name="Bar1"
select="/OfficeSales/Office[position()=1]/@sales"/>
```

```
<xsl:template name="CreateBars">
<xsl:variable name="Bar1"
select="/OfficeSales/Office[position()=1]/@sales"/>
<rect x="101" y="65" width="{0.4 * $Bar1}" height="20"
style="stroke:none; fill:#FF6600;" visibility="hidden">
<animate id="Anim1" attributeName="visibility" begin="3s" dur="5s"
  from="hidden" to="visible" fill="freeze"/>
</rect>
<rect x="101" y="165" width="{0.4 *
  (/OfficeSales/Office[position()=2]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;" opacity="0">
<animate id="Anim2" attributeName="opacity" begin="Anim1.end+2s"
  dur="5s" from="0" to="1" fill="freeze"/>
</rect>
<rect x="101" y="265" width="{0.4 *
(/OfficeSales/Office[position()=3]/@sales)}" height="20"
style="stroke:none; fill:#DDDDDD;">
<animate id="Anim3" attributeName="fill" begin="Anim2.end+2s" dur="5s"
  values="#DDDDDD; #FF0000; #00FF00; #FF6600" fill="freeze"/>
</rect>
<rect x="101" y="365" width="0" height="20"
style="stroke:none; fill:#FF6600;">
<animate id="Anim4" attributeName="width" begin="Anim3.end+2s" dur="5s"
  from="0"
to="{0.4 * (/OfficeSales/Office[position()=4]/@sales)}" fill="freeze"/>
</rect>
<rect x="101" y="465" width="{0.4 *
(/OfficeSales/Office[position()=5]/@sales)}" height="0"
style="stroke:none; fill:#FF6600;" visibility="hidden">
<animate id="Anim5" attributeName="height" begin="Anim4.end+2s" dur="5s"
  from="0" to="20" fill="freeze"/>
<set begin="Anim4.end+2s" from="hidden" to="visible"
  attributeName="visibility" fill="freeze"/>
</rect>
</xsl:template>
```

Listing 8.8 The CreateBars Template with SVG Animations Added (Part of SVGChartAnim.xsl).

```
<rect x="101" y="65" width="{0.4 * $Bar1}" height="20"
style="stroke:none; fill:#FF6600;" visibility="hidden">
<animate id="Anim1" attributeName="visibility" begin="3s" dur="5s"
  from="hidden" to="visible" fill="freeze"/>
</rect>
```

The <xsl:variable> element is unchanged. To add the animation, we need to split the <rect> element, which previously looked like the following code, into separate start and end tags.

```
<rect x="101" y="65" width="{0.4 * $Bar1}" height="20"
style="stroke:none; fill:#FF6600;"/>
```

We also add a visibility attribute to the <rect> element with the value of "hidden" so that the bar is not visible when the page loads. Between the start and end tag we insert an SMIL 2.0 <animate> element, which animates the visibility attribute from a value of "hidden" to a value of "visible" at the end of 5 seconds.

```
<animate id="Anim1" attributeName="visibility" begin="3s" dur="5s"
  from="hidden" to="visible" fill="freeze"/>
```

The attributes of the <animate> element control how the animation happens. The attributeName attribute defines which attribute on the <rect> element is to be animated. In this example, it is the visibility attribute. The begin attribute controls when the animation begins. The value of "3s" indicates that the animation begins 3 seconds after the page loads. The dur attribute controls the duration of the animation, which in this case is 5 seconds. Putting those two aspects of the animation together means that the bar will be altered from hidden to visible after 8 seconds, as indicated by the values of the from attribute and the to attribute. The fill attribute indicates that the appearance of the <rect> element will then be frozen in the visible state.

The attribute of the <animate> element (which I haven't mentioned yet) is the id attribute. As you will see in a moment, we use the value of the id attribute of the <animate> element to "chain" SVG animations. In other words, the animation of the second bar in the bar chart is linked to the animation with the id of "Anim1".

The animation for the San Francisco office depends on the following code:

```
<rect x="101" y="165" width="{0.4 *
(/OfficeSales/Office[position()=2]/@sales)}" height="20"
style="stroke:none; fill:#FF6600;" opacity="0">
<animate id="Anim2" attributeName="opacity" begin="Anim1.end+2s"
  dur="5s" from="0" to="1" fill="freeze"/>
</rect>
```

When the page loads the opacity attribute with a value of "0", this means that the bar is invisible. The begin attribute makes use of the id attribute on the earlier <animate> element. The San Francisco animation begins at "Anim1.end+2s", i.e., 2 seconds after the New York animation ends. Thus 2 seconds after the first animation is completed, the second animation starts.

NOTE **Be very careful how you write the content of the begin attribute when animations are being changed. It is important to avoid any whitespace between the characters that make up the value, or the animation won't work.**

The animation changes the opacity attribute of the bar from 0 to 1 over a period of 5 seconds, giving an impression that the bar slowly fades in. The fill attribute of the <animate> element ensures that the bar remains visible after completion of the animation.

The bar for the Amsterdam office depends on the following code:

```
<rect x="101" y="265" width="{0.4 *
(/OfficeSales/Office[position()=3]/@sales)}" height="20"
style="stroke:none; fill:#DDDDDD;">
<animate id="Anim3" attributeName="fill" begin="Anim2.end+2s" dur="5s"
  values="#DDDDDD; #FF0000; #00FF00; #FF6600" fill="freeze"/>
</rect>
```

It is fully visible when the page loads, but its color is exactly the same as the background color; therefore, we don't appreciate that we see it. The begin attribute shows that this animation starts 2 seconds after the previous one ends. The values attribute of the <animate> element means that over a period of 5 seconds the color changes from gray to red to green to orange. The fill attribute of the <animate> element freezes the fill color on the <rect> element to an orange color.

The animation for the London office is created using the following code:

```
<rect x="101" y="365" width="0" height="20"
style="stroke:none; fill:#FF6600;">
<animate id="Anim4" attributeName="width" begin="Anim3.end+2s" dur="5s"
  from="0"
to="{0.4 * (/OfficeSales/Office[position()=4]/@sales)}" fill="freeze"/>
</rect>
```

Again, the animation begins 2 seconds after the previous one ends. The bar is orange-colored but is not obvious when the page loads since its width is zero. The <animate> element changes the width of the bar from 0 to a value determined by the sales attribute in the source XML document. On screen the bar appears to grow in width progressively over 5 seconds.

The final animation uses the following code:

```
<rect x="101" y="465" width="{0.4 *
(/OfficeSales/Office[position()=5]/@sales)}" height="0"
style="stroke:none; fill:#FF6600;" visibility="hidden">
<animate id="Anim5" attributeName="height" begin="Anim4.end+2s" dur="5s"
  from="0" to="20" fill="freeze"/>
<set begin="Anim4.end+2s" from="hidden" to="visible"
  attributeName="visibility" fill="freeze"/>
</rect>
```

When the page loads, the height of the bar is set to zero, but I also had to set the visibility of the bar to "hidden", since otherwise the Adobe Viewer would display a very narrow bar of color. Thus after the fourth animation ends, there are two animations applied to the final bar for the Paris office. Visibility is changed from hidden to visible using the <set> element, which happens instantly 2 seconds after the fourth animation ends. Secondly, over a period of 5 seconds the height of the bar changes from 0 to 20.

The animations chosen are not necessarily those you would choose in a production environment, but they do serve to give you some indication of the animation capabilities of SVG that are possible without the use of a scripting language.

Figure 8.5 shows the animated chart partway through the series of chained animations.

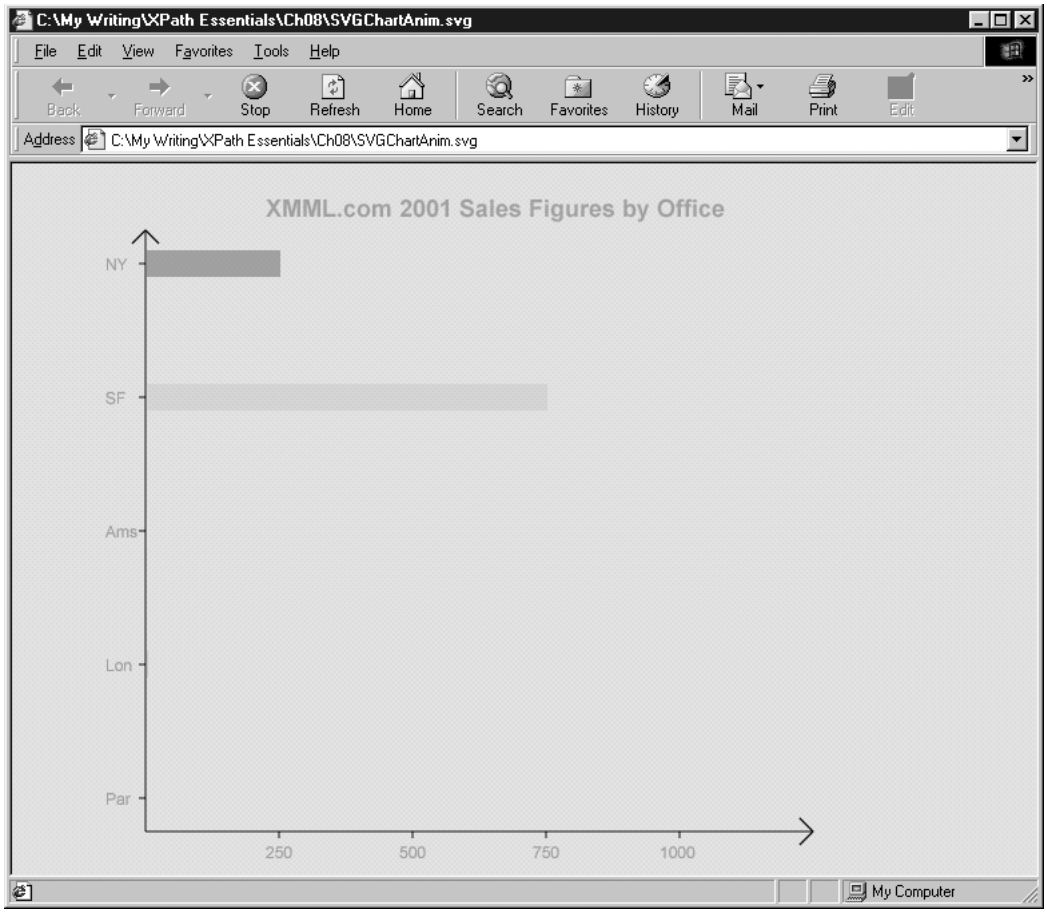


Figure 8.5 The Animated Bar Chart Partway through the Animation for the London Office.

Having seen how to create a bar chart using the SVG <rect> element, let's look at how to display the same data using a line chart that was created using the SVG <line> element.

Creating a Static SVG Line Chart

We could create a line chart using the skeleton that we used earlier but, for a line chart, it seems more natural to reverse the axes in the SVG chart, with the names of the offices along the horizontal axis and the values of sales along the vertical axis.

The following XSLT stylesheet (see Listing 8.9) creates the SVG line chart image with the use of the axes reversed. Most of the use of XPath is in the DrawLineGraph named template, which will be described next.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns="http://www.w3.org/2000/svg">

<xsl:output
  method="xml"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<svg width="100%" height="550px">
<title>XMML.com Sales by Office</title>
<rect x="0" y="0" width="100%" height="100%" style="fill:#DDDDDD"/>
<xsl:call-template name="CreateAxes"/>
<xsl:call-template name="DrawLineGraph"/>
<xsl:call-template name="CreateGraphTitle"/>
</svg>
</xsl:template>

<xsl:template name="CreateAxes">
<xsl:call-template name="CreateVerticalAxis"/>
<xsl:call-template name="CreateHorizontalAxis"/>
</xsl:template>

<xsl:template name="CreateVerticalAxis">
<line x1="100" y1="50" x2="100" y2="500" style="stroke:black; stroke-
  width:1;"/>
<line x1="90" y1="60" x2="100" y2="50" style="stroke:black; stroke-
  width:1;"/>
<line x1="110" y1="60" x2="100" y2="50" style="stroke:black; stroke-
  width:1;"/>
<!-- Create the axis marks for each office -->
<line x1="100" y1="100" x2="95" y2="100" style="stroke:black; stroke-
  width:1;"/>
<line x1="100" y1="200" x2="95" y2="200" style="stroke:black; stroke-
  width:1;"/>
<line x1="100" y1="300" x2="95" y2="300" style="stroke:black; stroke-
  width:1;"/>
<line x1="100" y1="400" x2="95" y2="400" style="stroke:black; stroke-
  width:1;"/>
<!-- Create the labels for the Y axis -->
<text x="69" y="100" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
1000
</text>
<text x="75" y="200" style="stroke:none; fill:#FF6600; font-size:12;
```

Listing 8.9 A Stylesheet to Create an SVG Line Chart (SVGLineChart.xsl).

```
font-family:Arial, sans-serif;">
750
</text>
<text x="75" y="300" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
500
</text>
<text x="75" y="400" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
250
</text>
</xsl:template>

<xsl:template name="CreateHorizontalAxis">
<line x1="100" y1="500" x2="700" y2="500" style="stroke:black; stroke-
  width:1;"/>
<line x1="690" y1="490" x2="700" y2="500" style="stroke:black; stroke-
  width:1;"/>
<line x1="690" y1="510" x2="700" y2="500" style="stroke:black; stroke-
  width:1;"/>
<!-- Create the axis marks for the sales value. -->
<line x1="200" y1="500" x2="200" y2="505" style="stroke:black; stroke-
  width:1;"/>
<line x1="300" y1="500" x2="300" y2="505" style="stroke:black; stroke-
  width:1;"/>
<line x1="400" y1="500" x2="400" y2="505" style="stroke:black; stroke-
  width:1;"/>
<line x1="500" y1="500" x2="500" y2="505" style="stroke:black; stroke-
  width:1;"/>
<line x1="600" y1="500" x2="600" y2="505" style="stroke:black; stroke-
  width:1;"/>
<!-- Create the labels for the horizontal axis. -->
<text x="190" y="520" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=1]/@abbreviation"/>
</text>
<text x="290" y="520" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=2]/@abbreviation"/>
</text>
<text x="390" y="520" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=3]/@abbreviation"/>
</text>
<text x="490" y="520" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=4]/@abbreviation"/>
</text>
```

Listing 8.9 (*Continued*) (*continues*)

```
<text x="590" y="520" style="stroke:none; fill:#FF6600; font-size:12;
font-family:Arial, sans-serif;">
<xsl:value-of select="/OfficeSales/Office[position()=5]/@abbreviation"/>
</text>
</xsl:template>

<xsl:template name="DrawLineGraph">
<line x1="200" y1="{500-0.4*(/OfficeSales/Office[position()=1]/@sales)}"
   x2="300" y2="{500-0.4*(/OfficeSales/Office[position()=2]/@sales)}"
   style="stroke:black; stroke-width:1;"/>
<line x1="300" y1="{500-0.4*(/OfficeSales/Office[position()=2]/@sales)}"
   x2="400" y2="{500-0.4*(/OfficeSales/Office[position()=3]/@sales)}"
   style="stroke:black; stroke-width:1;"/>
<line x1="400" y1="{500-0.4*(/OfficeSales/Office[position()=3]/@sales)}"
   x2="500" y2="{500-0.4*(/OfficeSales/Office[position()=4]/@sales)}"
   style="stroke:black; stroke-width:1;"/>
<line x1="500" y1="{500-0.4*(/OfficeSales/Office[position()=4]/@sales)}"
   x2="600" y2="{500-0.4*(/OfficeSales/Office[position()=5]/@sales)}"
   style="stroke:black; stroke-width:1;"/>
</xsl:template>

<xsl:template name="CreateGraphTitle">
<!-- Add the title for the graph -->
<text x="160" y="40" style="stroke:none; fill:#FF6600; font-size:18;
font-weight:bold; font-family:Arial, sans-serif;">
XMML.com <xsl:value-of select="/OfficeSales/@year"/> Sales Figures by
  Office
($000's)
</text>
</xsl:template>

</xsl:stylesheet>
```

Listing 8.9 (*Continued*)

The DrawLineGraph named template uses the values in the source XML document to define the start (x1 and y1 attributes) and/or end (x2 and y2 attributes) coordinates for each of the straight lines drawn in the graph.

```
<xsl:template name="DrawLineGraph">
<line x1="200" y1="{500-0.4*(/OfficeSales/Office[position()=1]/@sales)}"
   x2="300" y2="{500-0.4*(/OfficeSales/Office[position()=2]/@sales)}"
   style="stroke:black; fill:none; stroke-width=1;"/>
<line x1="300" y1="{500-0.4*(/OfficeSales/Office[position()=2]/@sales)}"
   x2="400" y2="{500-0.4*(/OfficeSales/Office[position()=3]/@sales)}"
   style="stroke:black; fill:none; stroke-width=1;"/>
```

```
<line x1="400" y1="{500-0.4*(/OfficeSales/Office[position()=3]/@sales)}"
   x2="500" y2="{500-0.4*(/OfficeSales/Office[position()=4]/@sales)}"
   style="stroke:black; fill:none; stroke-width:1;"/>
<line x1="500" y1="{500-0.4*(/OfficeSales/Office[position()=4]/@sales)}"
   x2="600" y2="{500-0.4*(/OfficeSales/Office[position()=5]/@sales)}"
   style="stroke:black; fill:none; stroke-width:1;"/>
</xsl:template>
```

In the first <line> element, for example, we use an attribute value template that incorporates the value of the sales attribute of the <Office> element child of the <OfficeSales> element, which is in position 1 in the calculation of the y1 attribute. In addition, we use the value of the sales attribute of the <Office> element in position 2 to define where the end of the <line> element is positioned.

```
<line x1="200" y1="{500-0.4*(/OfficeSales/Office[position()=1]/@sales)}"
   x2="300" y2="{500-0.4*(/OfficeSales/Office[position()=2]/@sales)}"
   style="stroke:black; fill:none; stroke-width:1;"/>
```

In a production environment you would be likely to use XSLT <xsl:variable> elements to hold the values to be used in calculating coordinate positions. The use of variable names within the attribute value templates would make reading easier, since the variable names are likely to be shorter.

A Weather Chart in a Scrolling SVG Text Window

SVG images can also provide dynamic display of text information either as an SVG image embedded in an HTML/XHTML Web page or within a standalone Web page.

NOTE **If you are unfamiliar with SVG as a Web page authoring tool, take a look at www.svgspider.com/default.svg, where several foundational techniques are demonstrated.**

Such SVG text displays could be used for updating share prices, weather information, etc. In the following example, I will show you how to use and display basic weather data within a scrolling SVG text window.

The XML source document is shown in Listing 8.10.

```
<?xml version='1.0'?>
<Weather>
<City name="London">
```

Listing 8.10 The Source Weather Data.

(*continues*)

```
<Temperature>67</Temperature>
<Moisture>Dry</Moisture>
</City>
<City name="New York">
<Temperature>
81
</Temperature>
<Moisture>Rain</Moisture>
</City>
<City name="Miami">
<Temperature>
90
</Temperature>
<Moisture>Dry</Moisture>
</City>
</Weather>
```

Listing 8.10 (*Continued*)

The XSLT stylesheet in Listing 8.11 creates a scrolling SVG window that displays weather information selected using XPath expressions from the XML source document in Listing 8.10.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xlink="http://www.w3.org/1999/xlink"
                xmlns="http://www.w3.org/2000/svg">

<xsl:template match="/">
<svg width="175px" height="250px">
<rect x="0" y="0" width="100%" height="100%" style="fill:#DDDDDD;
  stroke:black;
stroke-width:2;"/>

<text>
<tspan x="5" y="240" style="font-size:14; font-family:Arial, sans-serif;
stroke:#990066; fill:#990066">
<animate attributeName="y" begin="1s" dur="30s" from="240" to="-240"
repeatCount="indefinite"/>
XMML Weather Reports
</tspan>
```

Listing 8.11 The Stylesheet That Creates the Scrolling SVG Text Window. (SVGText-Weather.xsl)

```
<tspan x="5" dy="2em" style="font-size:10; font-family:Arial, sans-
  serif;">
XMML Weather Reports brings you
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
the latest international weather.
</tspan>
<tspan x="5" dy="2em" style="font-size:14; font-family:Arial, sans-serif;
fill:red;">
London
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
Temperature: <xsl:value-of
select="/Weather/City[@name='London']/Temperature"/>
F
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
serif;">
Moisture: <xsl:value-of select="/Weather/City[@name='London']/Moisture"/>
</tspan>
<tspan x="5" dy="2em" style="font-size:14; font-family:Arial, sans-serif;
fill:red;">
New York
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
Temperature: <xsl:value-of select="/Weather/City[@name='New
York']/Temperature"/>F
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
Moisture: <xsl:value-of select="/Weather/City[@name='New
York']/Moisture"/>
</tspan>
<tspan x="5" dy="2em" style="font-size:14; font-family:Arial, sans-serif;
fill:red;">
Miami
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
Temperature: <xsl:value-of
select="/Weather/City[@name='Miami']/Temperature"/>F
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
```

Listing 8.11 (*Continued*) (*continues*)

```
Moisture: <xsl:value-of select="/Weather/City[@name='Miami']/Moisture"/>
</tspan>
<tspan x="5" dy="2em" style="font-size:10; font-family:Arial, sans-
  serif;">
Further information is available
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
to subscribers at the XMML.com
</tspan>
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
Web site.
</tspan>
<a xlink:href="http://www.XMML.com/default.svg">
<tspan x="5" dy="2em" style="font-size:10; font-family:Arial, sans-serif;
fill:blue; stroke:blue">
http://www.XMML.com
</tspan>
</a>
</text>
</svg>
</xsl:template>

</xsl:stylesheet>
```

Listing 8.11 (*Continued*)

TEAMFLY

The XPath expressions that select element content are pretty straightforward. For example, the following code selects the content of the <Temperature> element for London, by means of a predicate on the <City> element, which defines the name attribute as having the value of "London".

```
<tspan x="5" dy="1em" style="font-size:10; font-family:Arial, sans-
  serif;">
Temperature: <xsl:value-of
select="/Weather/City[@name='London']/Temperature"/>
F
</tspan>
```

In this example, I have used the SVG <tspan> element nested within a <text> element. Notice that the <tspan> element has a dy attribute, which indicates the difference in y position of the <tspan> element in question from a preceding <text> or <tspan> element. This means that the content of the <tspan> scrolls upward as the preceding <text> or <tspan> element scrolls.

Figure 8.7 shows the animated weather report text partway through its animation.

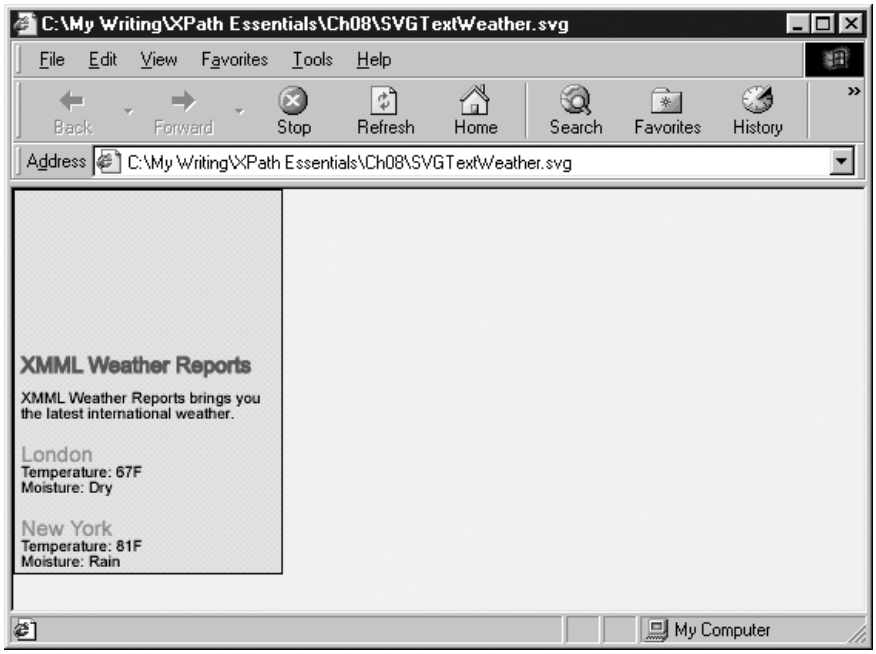


Figure 8.7 Weather Report Text Partway through its Animation.

Limitations of Transformations to Produce SVG

The examples I have shown you in the earlier part of this chapter make use of XPath 1.0 and XSLT 1.0. XPath provides a number of relatively simple number functions as shown in the following list:

- number ( )
- sum ( )
- floor ( )
- ceiling ( )
- round ( )

To those, XSLT 1.0 adds only the format-number ( ) function. Thus with XPath 1.0 and XSLT 1.0, we have a very limited armory of numerical functions. To those we can, of course, add such node-set functions as the count ( ) function.

Thus, with this limited range of functions, we can fairly readily produce bar charts and line graphs, but the lack of more numerical, particularly trigonometric functions in XPath or XSLT is a significant limitation as to the variety of graphical presentations in SVG that XPath and XSLT can produce.

The W3C has issued Requirements Working Drafts for both XPath 2.0 and XSLT 2.0. The XPath Requirements Working Draft is located at www.w3.org/TR/xpath20req and the XSLT 2.0 Requirements Working Draft is located at www.w3.org/TR/xslt20req. There is no mention in either requirements document of a need for additional numerical functions in XPath or XSLT. Thus, it seems likely that for some considerable time the use of XPath and XSLT will be limited to the creation of line and bar graphs.

It is possible that one or more of the XSLT processors will implement a useful range of numerical extension functions to assist creation of SVG using XSLT. Since such creation of SVG is likely to be done server side, the lack of portability of such extension functions is unlikely to be a significant concern.

Until the functionality provided by XPath and XSLT numerical functions is improved, if you want to create SVG dynamically and line charts and bar charts do not meet your presentation needs, you would probably be better advised to consider using a programming language with a more complete range of numerical processing functionality.

An alternative approach is to use graphing tools such as those found in Microsoft Excel, together with a simple "printing" utility such as SVG Maker, which prints SVG output to a file. Further information on SVG Maker is located at www.svgmaker.com. Other standalone tools to generate SVG graphs include the Corda Pop Chart Image Server Pro, which can produce an extensive range of SVG graphs. Further information is located at www.corda.com.

Looking Ahead

In this chapter I have been able to give you only a glimpse of SVG's potential to display information contained in XML source documents. By using XPath expressions to appropriately select parts of the XML source data, attractive and informative SVG images can be created.

Chapter 9 will address how XPath can be used with XPointer.

Using XPath in XPointer

This chapter will introduce you to the use of XPath with the XML Pointer Language XPointer and to the extensions to XPath contained in XPointer. At the time of writing, XPointer remains a Working Draft at the W3C; therefore, details of what is described in this chapter may be particularly subject to change.

There has been some public dispute within the committee developing XPointer as to whether a simplified version of XPointer should be the basis for the initial XPointer recommendation or whether the version in the previous public Working Drafts should form the basis for further development of Xpointer. It is possible that significant parts of the functionality described in this chapter may be discarded, at least in XPointer version 1.0. I suggest that if you want to implement XPointer, you check the latest version and pay particular attention to any change history information.

NOTE The last call XPointer Working Draft of January 8, 2001, is located at www.w3.org/TR/2001/WD-xptr-20010108. Any future versions of the XPointer specification will be located at www.w3.org/TR/xptr. If that URL still shows the January 2001 Working Draft, then no further revision of the XPointer specification has taken place since this chapter was written.

Let's first look at what XPointer actually is.

What Is XPointer?

XML Pointer, XPointer, is a language that is intended to allow the addressing of document fragments. Specifically, XPointer is intended by W3C to be the fragment identifier language for documents with media type of `xml/text`, `application/xml`, `text/xml-external-parsed-entity` or `application/xml-external-parsed-entity`. This means that XPointer is, or, more precisely, will be the W3C-approved language to address fragments in XML documents.

XPointer can address parts of XML documents such as elements and character strings, whether or not they possess an ID attribute. XPointer makes use of such XML document properties as element types, attribute values, or relative position. In this way, it is very similar to XPath, but its capabilities to address fragments of XML documents go beyond those in XPath.

XPointer is built on XPath, with the following extensions:

- The ability to address *points* and *ranges* as well as whole nodes
- The ability to locate information by string matching
- The ability to use addressing expressions in URI references as fragment identifiers

XPointer does not define a way to address within a DTD or within the XML declaration and, in that respect, resembles XPath.

One intended use for XPointer is in defining the targets of links related to XML documents. Thus XPointer will be frequently used with the linking technology described in the XLink Recommendation, once the specification is finalized. The XLink Recommendation is available online at www.w3.org/TR/xlink/ and has been a full W3C Recommendation since June 27, 2001.

In order to understand the kind of role that XPointer, particularly when used with XLink, is likely to be able to provide, let's first take a look backward at HTML fragment identifiers and examine their capabilities and shortcomings.

HTML Fragment Identifiers

HTML makes extensive use of hyperlinking by means of the <A> element. The <A> element has an HREF attribute, which specifies the location of a resource that is being linked to. HTML or, more precisely, Uniform Resource Identifiers (URIs), which are used so frequently in association with HTML, makes use of the "#" character to indicate a fragment identifier.

Let's take a look at how HTML fragment identifiers are used. For simplicity and brevity we will simply link to documents in a single directory. The same technique works in a similar fashion with full URIs. A sample listing is shown in Listing 9.1.

Notice that I have included a substantial number of <BR> tags to separate out the brief paragraphs in Anchors.html, which lets us mimic a very long HTML document without taking up pages with lots of text. Without such separation of the parts of the HTML document, we couldn't be sure which part was being linked to, since the browser would not be required to scroll to display the desired anchor.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>HTML Anchors Demo</TITLE>
</HEAD>
<BODY>
<H3>This is a demonstration of using fragment identifiers in HTML.</H3>
<A NAME="First"></A>
<P>This is a paragraph which follows the anchor named "First".<P>
<BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR>
<A NAME="Second"></A>
<P>This is a second paragraph which uses the anchor named "Second".</P>
<BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR><BR>
<A NAME="Third"></A>
<P>This is a third paragraph which uses the anchor named "Third".</P>
<BR><BR><BR><BR><BR>
<A NAME="Fourth"></A>
<P>This is a fourth paragraph which uses the anchor named "Fourth".</P>
<BR><BR><BR><BR><BR>
<P>This is a fifth paragraph which has no anchor and so neither we nor
  anyone
else can link to it.</P>
<BR><BR><BR><BR><BR>
</BODY>
</HTML>
```

Listing 9.1 An HTML Document to Demonstrate Anchors (Anchors.html).

Notice also that we have to explicitly create named anchors if we, or anyone else, are to be able to link to a particular spot in the page. The fifth <P> tag is something we can't link to since there is no named anchor.

We can create a simple HTML document to link to Anchors.html, as shown in Listing 9.2.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>This Links to Anchors</TITLE>
</HEAD>
<BODY>
<H3>This file allows you to select which anchor you want to link to in
```

Listing 9.2 An HTML Document to Link to the Anchors in Listing 9.1 (LinksToAnchors.html). (*continues*)

```
  Anchors.html.</H3>
<A HREF="Anchors.html#First">First</A><BR>
<A HREF="Anchors.html#Second">Second</A><BR>
<A HREF="Anchors.html#Third">Third</A><BR>
<A HREF="Anchors.html#Fourth">Fourth</A><BR>
<A HREF="Anchors.html#Fifth">Fifth</A><BR>
</BODY>
</HTML>
```

Listing 9.2 (*Continued*)

If we open LinksToAnchors.html, Listing 9.2, in a browser, it looks like Figure 9.1.

If we select, for example, the link to the third anchor, we get the visual appearance shown in Figure 9.2. We have been able to link directly to the specific desired point in the HTML document—it is displayed at the top of the browser window where it is most likely to immediately attract our attention.

However, if we attempt to link to the fifth paragraph, which has no <A> element to create a named anchor, then we find that we simply link to the top of Anchors.html. In a lengthy target document this could leave users thoroughly confused as to what the point of the hyperlink was. They might, after some searching, be able to locate the desired part of the target document, particularly if there was some verbal cue in the document it was being linked from. But it is much less efficient and reliable than a link directly to the part of the target document that is of interest.

Since the browser could not interpret the fragment "fifth", which we asked it to locate, it took us to the beginning of the document, as you can see in Figure 9.3. While this

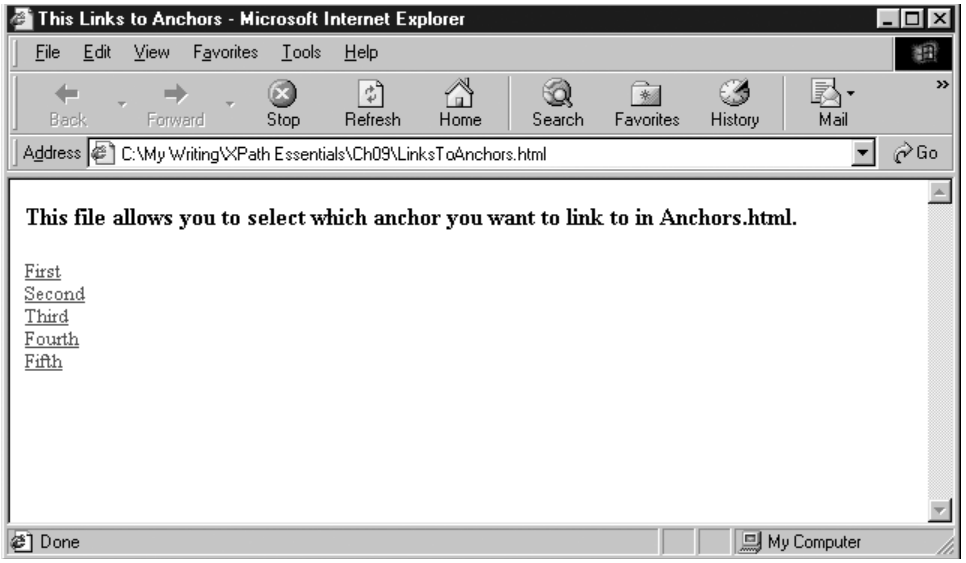


Figure 9.1 Links to the Anchors in Anchors.html.

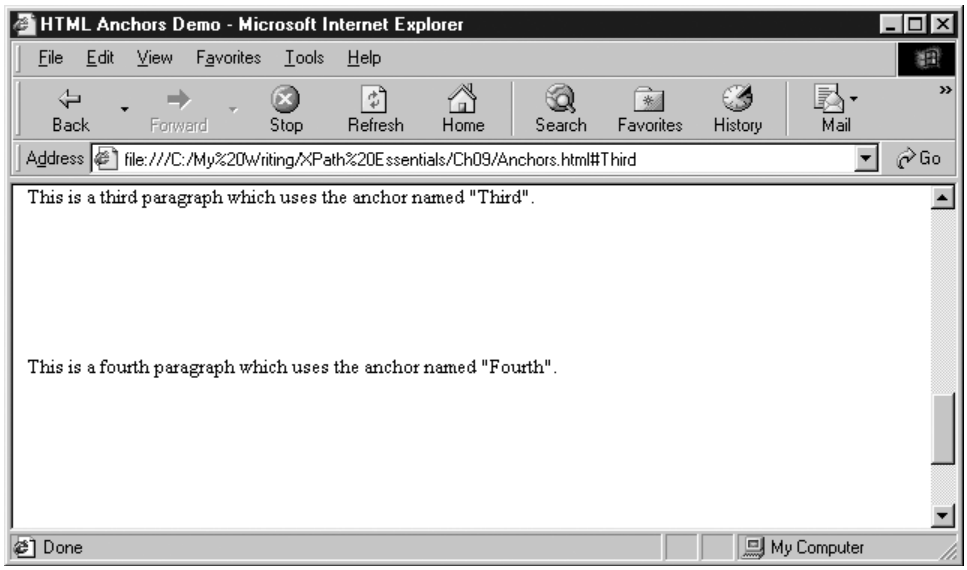

HTML Anchors Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address file:///C:/My%20Writing/XPath%20Essentials/Ch09/Anchors.html#Third
Go
This is a third paragraph which uses the anchor named "Third".
This is a fourth paragraph which uses the anchor named "Fourth".
Done
My Computer

Figure 9.2 Linking to a Predefined Anchor.

is reasonable fallback behavior by the browser, it would not solve the problem of locating a specific piece of information in what might be a very lengthy document.

There is a need to have anchor points already declared by the HTML document author. Without HTML anchors it is impossible to take a reader directly to a desired bit of text unless you, or someone with whom you are cooperating, have provided anchors to

HTML Anchors Demo - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address file:///C:/My%20Writing/XPath%20Essentials/Ch09/Anchors.html#Fifth
Go
This is a demonstration of using fragment identifiers in HTML.
This is a paragraph which follows the anchor named "First".
Done
My Computer

Figure 9.3 Linking to the Beginning of a Document When a Fragment Identifier Cannot Be Located.

the fragment of the HTML document that is of interest to you or to your readers. Think of an online HTML-based teaching tool that needs to illustrate certain points from online documents to which it has read-only access. There is no way that the student can be shown immediately and precisely what the teacher had in mind. The student can be taken to the document, but not to the section of particular interest.

A linking technology that offered the capacity to link to fragments in documents to which the author of the link did not have write access offers more efficient and more targeted online learning. A technology such as XPointer is expected to offer precisely that advantage.

A further limitation of HTML fragment identifiers is that the entire document being linked to is loaded, not just the section of interest. This is a disadvantage, particularly in very lengthy documents where named anchors are potentially of most use. The capacity to selectively retrieve a desired fragment, which XPointer will provide, would avoid unnecessary use of bandwidth, for example.

Having looked briefly at limitations of HTML fragment identifiers, let's look at why XPath alone does not provide an answer to those issues.

What XPath Cannot Express

If we have an XML source document like the one shown in Listing 9.3, and want to select a part of that document as illustrated in Figure 9.3, we will find that XPath simply cannot represent the selection indicated in Figure 9.4. It does not easily translate into something that a data model, like the XPath data model, which incorporates nodes alone can represent.

It is not the purpose of this example to indicate that selections may only be made visually. I simply want to indicate that it is possible to select, either visually or programmatically, parts of an XML document that XPath cannot easily describe or select. Thus we need a technology that goes beyond the capabilities that XPath has. Yet that technology, XPointer, can make use of and build on XPath's concepts of nodes and add further concepts of its own.

Let's examine XPointer terminology and concepts so we can better understand what it is that XPointer does.

```
<?xml version='1.0'?>
<XPointerDemo>
<Document>
<Paragraph number="1">Here is something in place of the first
paragraph.And a reader is
likely to find this sentence very intersting.</Paragraph>
<Paragraph number="2">And this sentence is interesting too. But not this
  one, unfortunately.</Paragraph>
</Document>
</XPointerDemo>
```

Listing 9.3 A Source Document to Demonstrate Xpointer (XPointerDemo.xml).

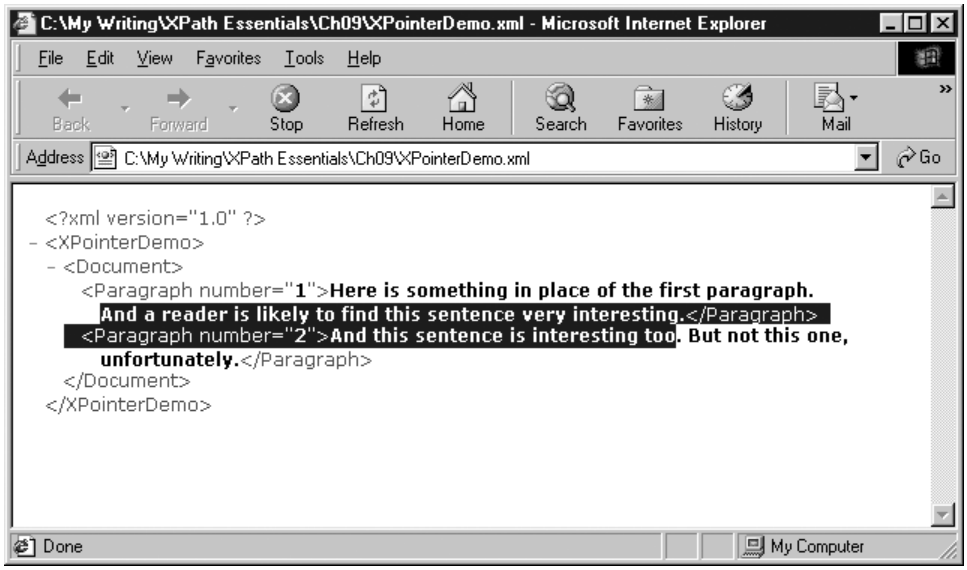


Figure 9.4 A Selection Made from XPointerDemo.xml.

Understanding XPointer

XPointer has many similarities to XPath; it is built on top of XPath as I have indicated earlier in this book. However, XPointer also has similarities in its data model to the Document Object Model and also, in its use of ordered lists, has similarities to the XML Information Set.

In order to help you understand a little about XPointer we will first look at the terminology that it adds to the notions we are already familiar with in XPath.

XPointer Terminology

XPointer adds several terms that extend the notion of a node present in XPath and adds concepts from the Document Object Model Level 2. It uses, in a sense, a hybrid of the data model of DOM2 and of XPath.

NOTE The DOM Level 2 Recommendations that are particularly relevant background to the XPointer specification are DOM Level 2 Core (located at www.w3.org/TR/DOM-Level-2-Core) and the DOM Level 2 Traversal-Range Recommendation (located at www.w3.org/TR/DOM-Level-2-Traversal-Range).

XPointer adds the term *location* to refer to a generalization of the notion of a node. A location can include nodes of the types defined in XPath, but also includes the concepts of a *point* and a *range*, which will be described later in this chapter.

The selected piece of text, shown earlier in Figure 9.4, would constitute a *range*. The position at the beginning of the highlighted area would correspond to a *point*, as would the position at the end of the highlighted area of text.

The XPointer specification defines a *point* as "a position in XML information," which seems less than informative, at least if you are not familiar with the background in DOM Level 2. The DOM Level 2 description of the notion of position is in Chapter 2.2.1 of the Traversal-Range module referred to above. This is potentially confusing terminology, since what XPointer calls a point is what DOM Level 2 refers to as a position. The reason that the term point was chosen in XPointer rather than position was because of the potential for confusion with the XPath notion of context position and the XPath position ( ) function.

An XPointer range is all the information in an XML document between a pair of end points.

The XPointer notion of a location, therefore, includes nodes, ranges, and points. An XPointer expression or function may return a *location-set*, which is an ordered list of locations. A location-set corresponds to the XPath notion of a node set but also includes ranges and points within the location-set.

If we had the following source code

```
<Paragraph ID="Warning3">
<Sentence>This is a specific warning.</Sentence>
<Sentence>The warning consists of three sentences.</Sentence>
<Sentence>This is the final sentence.</Sentence>
</Paragraph>
```

and assuming that the ID attribute has been declared as an ID attribute in the relevant DTD, we could use the value of the ID attribute of the <Paragraph> element to construct an XPointer, which would return a location-set containing three locations—the `Sentence` children in the list that makes up the location-set.

```
xpointer(id('Warning3')/Sentence)
```

XPointer uses the term *singleton* to refer to a location-set that consists of a single location. A range and a point are always singletons.

The following XPointer would form a singleton (since it is a range), although it refers to the same locations as those described in the previous XPointer expression.

```
xpointer(id('Warning3')/Sentence[1]/range-to(following-
  sibling::Sentence[3])
```

XPointer also uses the term *sub-resource* to refer to that part of an XML document that is referred to by an XPointer. In this context, a resource is likely to be the whole XML document but the sub-resource might be an element node, a range, a point, or some other portion of the XML document referred to by the XPointer.

NOTE An XPointer conforms to the XPointer specification if it uses the syntax defined in the XPointer specification. It need not point to a sub-resource that actually exists at any point in time. This approach makes sense since,

depending on how a team of programmers are working together, the person developing the XPointers might have completed development at a time when the documents, or fragments of them, to which those XPointers are to be applied are not completely written or populated.

You may find it helpful to see XPath and XPointer terminology side by side, in order to see more clearly the corresponding terms in each language. Those are shown in Table 9.1.

The XPointer Data Model

The XPath data model, derived from a source XML document, is augmented by XPointer. XPointers and XPath location paths operate by selecting parts of the data set constructed from the source XML document, often by reference to relationship to other parts of the data set. XPointers make iterative selections, with each selection operating on the previously selected one. This corresponds to the notion of a location step in XPath location paths, where the node set selected by one location step is the input to any following location step.

The iterative approach is illustrated by the earlier code snippet

```
xpointer(id('Warning3')/Sentence)
```

The XPointer first selects a location that possesses the ID attribute "Warning3", and then for that selected location-set determines whether it has any child locations with the name "Sentence". In the example code snippet we looked at earlier, there were three such child locations, each of which was an element node.

In XPointer, selection of parts of the hierarchical information tree is made using axes, predicates, and functions. XPointer uses XPath axes. Predicates apply further tests to the location-set selected by the relevant axis. In addition to the XPath functions described in Chapter 5, XPointer also provides several additional functions, which are described later in this chapter.

Table 9.1 Comparison of XPath and XPointer Terms

XPATH	XPOINTER
Node	Location
Node Type	Location type
Node Set	Location-set
—	Ranges
—	Point
Axis	Axis
Node test	Test
Context	Evaluation Context

Point Locations

The point location is a location type that is not recognized as an XPath node type. A point location defines one end of a range location.

NOTE **XPointer borrows the notion of a point from DOM; however, the character sets of DOM and XPath and XPointer differ. DOM uses UTF-16. XPath and XPointer use UCS characters.**

More formally, a point is defined within the XPointer model of an XML document by a *container node*, which is the parent node of the point, and a non-negative integer called the *index*. A point can represent a position preceding any character or preceding or following any node in the XPointer representation of an XML document.

The self axis of a point is the point itself. The parent axis of the point is a location-set containing a single location, the container node.

The XPointer specification adds a further term, a *node-point*. A node-point is a point that has as its parent a node that can have child nodes (that is, the root node or element nodes). In that situation the index is an index into the child nodes. The index of a node-point must be greater than or equal to zero and less than or equal to the number of child nodes of the container node. An index of zero indicates the point immediately before any child nodes and an index of "n" indicates the point immediately after the *n*th child node.

NOTE **The index, which refers to points, is zero-based. This is in contrast with the one-based counting of XPath functions, such as the position ( ) function and the one-based counting of the XPointer string-range ( ) function (discussed later).**

There is a further term, a *character-point*, which exists when the container node is not of a node type that can have children. This occurs, for example, when the container node is a text node, but also occurs when the container node is a comment node or processing instruction node. In that situation the index is an index into the string-value of the container node. The index at a character-point must be greater than or equal to zero and less than or equal to the length of the string-value of the node. An index of zero indicates a character-point immediately before the first character of the string-value of the node and a non-zero index of "n" indicates a character-point immediately after the *n*th character of the string-value of the container node.

A point location does not have an expanded-name. The string-value of a point is empty.

The axes of a point location are defined as follows:

- The child, descendant, preceding-sibling, following-sibling attribute, and namespace axes are empty.
- The self axis contains the point itself.
- The parent axis contains the container node of a node-point.

- The ancestor axis contains the container node of a node-point as well as the ancestor nodes, if any, of the container node.
- The siblings of a node-point are the children of the container node, which occur before and after the node-point.

The current draft version of the XPointer specification does not define the content of the parent or ancestor axes for a character-point.

Range Locations

A location of type *range* is defined by two points: a *start point* and an *end point*. A range represents all the content and structure of the XML document between the start point and the end point. The start point and the end point must be contained within the same document, and the start point must not occur after the end point in document order.

A *collapsed range* is a range whose start point and end point are equal.

The XPointer specification defines the allowed ranges as follows: "If the container node of one point of a range is a node of a type other than element, text, or root, the container node of the other point of the range must be the same node. For example, it is allowed to specify a range from the start of a processing instruction to the end of an element (encompassing both those nodes, but still a singleton), but not to specify a range from text inside a processing instruction to text outside it."

A range location does not have an expanded-name.

The string-value of a range may be defined in either of two ways, depending on the circumstances. If the start point and the end point are both character points and both the start point and the end point container nodes are the same node, then the string-value consists of the characters between the two points. Otherwise, the string-value consists of any characters between the start point and the end point that are in text nodes.

The axes of a range location are the same as the axes of its start point.

A *covering range* is a range that completely encompasses a location. The detailed definition of a covering range depends on the location with which it is associated:

- When the location is a range location, the covering range is the same as the range.
- When the location is an attribute location or a namespace location, the container node of the start point and end point of the covering range is the attribute location or namespace location. The index of the start point of the covering range is zero, and the index of the end point of the covering range is the length of the string-value of the attribute location or namespace location.
- When the location is the root location, the container node of the start point and end point of the covering range is the root node. The index of the start point of the covering range is zero and the index of the end point of the covering range is the number of children of the root location.
- When the location is a point location, the start point and end point of the covering range are the point itself.

- When the location is any other location type, the container node of the start point and end point of the covering range is the parent of the location. The index of the start point of the covering range is the number of preceding sibling nodes of the location, and the index of the end point is one greater than the index of the start point.

Character Escaping

XPointer is designed so that it can be used in conjunction with XLink. Therefore, it must be able to be used in URIs. URIs require escaping of certain characters. Similarly, the existence of XPointers in XML documents means that there can be expected to be additional XML-related character escaping requirements. Further detail is provided in Chapter 4.1 of the XPointer Working Draft.

XPointer's Three Syntaxes

Just as XPath has four different syntaxes, so XPointer has three. The *full XPointers* are the unabbreviated form of the XPointer syntax. The two more abbreviated forms of syntax are the *bare names* and *child sequences*.

We will use an XML source document, shown in Listing 9.4, to briefly illustrate the available forms of XPath syntax.

Notice that I have added some id attributes to the XML document. It is likely that many documents would not possess ID attributes that we could conveniently use. While

```
<?xml version='1.0'?>
<Book>
<Preface>This is the preface.</Preface>
<Introduction>Let me introduce you to XPointer.</Introduction>
<Chapters id="AB">
<Chapter id="AB001" number="1">The first chapter about XPointer</Chapter>
<Chapter id="AB002" number="2">The second exciting chapter about
  XPointer.</Chapter>
<Chapter id="AB003" number="3">The third chapter exploring
  XPointer.</Chapter>
<Chapter id="AB004" number="4">The fourth chapter examining
  XPointer.</Chapter>
<Chapter id="AB005" number="5">The fifth chapter.</Chapter>
</Chapters>
<Appendices>
<Appendix number="1">Appendix about XPointer functions.</Appendix>
<Appendix number="2">XPointer Glossary</Appendix>
</Appendices>
</Book>
```

Listing 9.4 A Description of a Book in XML (Book.xml).

XPointer can make use of ID attributes when they are present, it also needs a form of syntax, which can navigate to document fragments in the absence of both suitable ID attributes and write access to the document of interest.

First let's look at the full XPointer syntax.

Full XPointers

A full XPointer will always have the following code as the basis for an XPointer expression. The other components of the XPointer are contained within the parentheses.

```
xpointer( )
```

Let's suppose we wanted to select the location for the <Chapter> element that possesses the id attribute with value of "AB002". We can do that using the following XPointer:

```
xpointer(id('AB002'))
```

Bare Names

The bare names syntax has been provided to encourage the use of ID attributes, since that is considered a relatively stable part of document structure, likely to survive changes that might make the desired location inaccessible by, for example, child sequences (to be discussed in the next section). A second purpose of the XPointer bare names syntax is to provide similar behavior to the HTML fragment identifier (discussed earlier in this chapter) for XML documents where such behavior would be expected and useful.

Bare names provide an abbreviated syntax form that can substitute more succinctly for full XPointers. For example, if we had the following full Xpointer, the location representing the <Chapters> element would be selected.

```
xpointer(id("AB")
```

The bare names syntax can express that much more succinctly as

```
#AB
```

Just as accessing an HTML anchor requires write access to an HTML document, so it is necessary to have, or make use of, write access to the XML document in question. If ID attributes have not been included in the target document, bare names cannot be used.

Child Sequences

A child sequence selects an element by a series of steps separated by the "/" character. For example, if we wished to select the third <Chapter> element in our example document, we could use the following child sequence:

```
/1/3/3
```

The first "/" character represents the root node, as in XPath. The "1" in the first step indicates the first element child of the root node, which, in this example, is the location representing the <Book> element. The third child location of the location representing the <Book> element is the location representing the <Chapters> element. Its third child is the location we want to select.

The child sequence syntax is clearly very succinct, but is also very poorly expressed. There is no way to extract the meaning of the child sequence without being intimately familiar with the structure of the document to which it is being applied.

A further disadvantage of the child sequence approach is that it is very vulnerable to slight changes in document structure. Suppose, for example, that we modified the source document as shown in Listing 9.5 by adding a simple XML comment.

Does our child sequence now return the desired location? The answer is a resounding no. The <Acknowledgment>element that we have just added to the document is now the second child of the <book>element. Thus an XPointer processor will attempt to look for the third child of the <book>element and will, of course, find the <Introduction> element. It will then try to look for the third element child of the <Introduction>element and fail to find any such child.

A full XPointer is considerably more resistant to such changes in document structure. To select the third <Chapter> element location using a full XPointer, we would have used the following code which would have worked nicely with the original form of the source document and would have selected the same location with the modified form of the document as well.

```
xpointer(id("AB003"))
```

```
<?xml version='1.0'?>
<Book>
Acknowledgment>Thanks to the tech editor!/Acknowledgment>
<Preface>This is the preface.</Preface>
<Introduction>Let me introduce you to XPointer.</Introduction>
<Chapters id="AB">
<Chapter id="AB001" number="1">The first chapter about XPointer</Chapter>
<Chapter id="AB002" number="2">The second exciting chapter about
  XPointer.</Chapter>
<Chapter id="AB003" number="3">The third chapter exploring
  XPointer.</Chapter>
<Chapter id="AB004" number="4">The fourth chapter examining
  XPointer.</Chapter>
<Chapter id="AB005" number="5">The fifth chapter.</Chapter>
</Chapters>
<Appendices>
<Appendix number="1">Appendix about XPointer functions.</Appendix>
<Appendix number="2">XPointer Glossary</Appendix>
</Appendices>
</Book>
```

Listing 9.5 The XML Document from Listing 9.4 with a Comment Inserted (Book02.xml).

XPointer's Two Schemes

The code examples I have used so far use one of XPointer's two *schemes*. The two schemes in XPointer are *xpointer* and *xmlns*. The XPointer specification envisages that further schemes may be added in a future version of XPointer and reserves for future use all schemes when the media type is text/xml, application/xml, text/xml-external-parsed-entity, or application/xml-external-parsed-entity.

The xpointer Scheme

The xpointer scheme may be used in any XPointer. Its more straightforward use is within an XML document, where it can be used alone, as you have seen in the earlier examples. However, there are times when the xpointer scheme must be used in conjunction with the xmlns scheme.

The xmlns Schem

In order to make the purpose of the xmlns scheme clear, let's first take a look at a slightly contrived source document. Listing 9.6 uses one namespace prefix to refer in the same document to two different namespace URIs, which is quite legal as far as XML is concerned.

If we then have an XPointer like the following, it isn't possible for the XPointer processor to distinguish which of the two different elements, <XMML:Section>, is being referred to since no namespaces are in scope.

```
<?xml version='1.0'?>
<Document>
<XMML:Document xmlns:XMML="http://www.XMML.com/Documentation/">
<XMML:Section>Something here.</XMML:Section>
<XMML:Section>Something else here.</XMML:Section>
<XMML:Section>Yet something different here.</XMML:Section>
<XMML:Invoice xmlns:XMML="http://www.XMML.com/Finance/">
<XMML:Section type="Address">
<!-- Address information goes here. -->
</XMML:Section>
<XMML:Section type="LineItems">
<!-- Information about line items goes here. -->
</XMML:Section>
</XMML:Invoice>
<XMML:Summary>
</XMML:Summary>
</XMML:Document>
<Document>
```

Listing 9.6 An XML Document Where One Namespace Prefix Refers to Two Namespace URIs (xmlns.xml).

```
xpointer(//XMML:Section)
```

A sub-resource error would result. It is to address potential difficulties like this that the xmlns scheme exists.

Let's look at how the xmlns scheme allows us to remove the ambiguity that is present in the earlier code. The following code removes the ambiguity.

```
xmlns(XMML=http://www.XMML.com/Documentation/) xpointer(//XMML:Section)
```

This code uses the namespace prefix XMML to refer to the namespace URI "http://www.XMML.com/Documentation/" and to evaluate the XPointer `xpointer (//XMML:Section)`. In our example document, this would select a location-set containing three locations, each referring to an element node that is in the namespace http://www.XMML.com/Documentation/.

If we wanted to address the <XMML:Section> elements nested within the <XMML:Invoice> element (those in the http://www.XMML.com/Finance/namespace), we could use the following XPointer:

```
xmlns (XMML=http://www.XMML.com/Finance/)
xpointer(//XMML:Section)
```

The XPointer specification allows you to use a different namespace prefix in either or both of the uses of the xmlns scheme above. Since the XPointer processor uses the namespace URI rather than the namespace prefix, this will cause no difficulty for the processor, but might confuse a casual reader.

XPointer's Role

It is likely that XPointer, just as XPath, will be a technology used by or with other XML-related technologies.

Perhaps the most obvious major role for XPointer is in combination with XLink as the XML means to express a relationship between two resources or sub-resources.

It is likely that other XML-oriented applications will use XPointer to make use of the fragments identified by an XPointer. For example, an XML document processing application might use an XPointer to scroll a document window to the beginning of the relevant part of the requested document and highlight all the text that is contained in the range.

XPointer Functions

XPointer can make use of XPath functions, although XPath functions have no awareness of either ranges or points. Thus XPointer requires additional functions in order to appropriately process ranges and points. These are described briefly in the following sections.

NOTE **I have presented the XPointer functions in alphabetical order. With the exception of the here ( ) and origin ( ) function, the XPointer functions all relate to XPointer ranges.**

end-point ( ) Function

The end-point ( ) function takes a location-set argument and returns a location-set.

For each location in the argument location-set, the end-point ( ) function adds a point location to the result location-set. The point that is added to the result location-set represents the end point of the relevant location in the argument location-set.

here ( ) Function

The here ( ) function takes no arguments and returns a location-set.

The here ( ) function returns a location-set with a single member. The member of the location-set may be determined in one of two ways:

- If the XPointer being evaluated appears in a text node inside an element node, the location returned by the here ( ) function is the element node.
- Otherwise, the location returned is the node that directly contains the XPointer being evaluated.

For example, if we had the following code snippet as part of an XML document, the XPointer would find the first occurring Section node in the ancestor axis (the Section node representing the <Section> element with number attribute of value "5"), and would then select the first Section element node in the preceding axis (the Section element node representing the <Section> element with number attribute of value "4").

```
<Section number="3">Something here</Section>
<Section number="4">Something else here.</Section>
<Section number="5">
<Button>
xlink:type="simple"
xlink:href="#xpointer(here()/ancestor::Section[1]/preceding::Section[1]"
>Click here for Section 4</Button>The section text goes here.</Section>
```

origin ( ) Function

The origin ( ) function takes no arguments and returns a location-set.

The origin ( ) function enables addressing relating to third party or inbound XLink links. The origin ( ) function allows XPointer to express relative locations when links do not reside directly at one of their end points. The origin ( ) function returns a location-set with a single member, which locates the element from which a user or application initiated traversal of the XLink link.

range ( ) Function

The range ( ) function takes a location-set argument and returns a location-set.

The range ( ) function returns ranges covering the locations in the argument location-set. For each location in the argument location-set, the range ( ) function representing the covering range of the argument location is added to the result location-set.

range-inside ( ) Function

The range-inside ( ) function takes a location-set argument and returns a location-set.

The XPointer specification describes the range-inside ( ) function as follows, "The `range-inside` function returns ranges covering the contents of the locations in the argument location-set. For each location *x* in the argument location-set, a range location is added to the result location-set. If *x* is a range location or a point, then *x* is added to the result location-set. If *x* is not a range location, then *x* is used as the container node of the start and end points of the range location to be added; the index of the start point of the range is zero; if the end point is a character point, then its index is the length of the string-value of *x*, and otherwise is the number of children of *x*."

range-to ( ) Function

The range-to ( ) function takes a location-set as its argument and returns a location set.

For each location in the context, the range-to ( ) function returns a range. The start of the range is the start point of the context location and the end of the range is the end point of the location found by evaluating the argument to the range-to ( ) function with respect to that context location.

If we had the following code,

```
<Chapter id="Chap02">Some text here.</Chapter>
<Chapter id="Chap03">Some more text here.</Chapter>
<Chapter id="Chap04">Yet more text here.</Chapter>
```

we could use the range-to ( ) function as follows to define a range that encompasses the first two <Chapter> elements shown above.

```
xpointer(id("Chap02")/range-to(id("Chap03")))
```

TEAMFLY

start-point ( ) Function

The start-point ( ) function takes a location-set argument and returns a location-set.

For each location in the argument location-set, the start-point ( ) function adds a point location to the result location-set. The point that is added to the result location-set represents the start point of the relevant location in the argument location-set.

string-range ( ) Function

The string-range ( ) function returns a location-set. It takes a location-set argument and a string argument and optional number arguments.

The following code returns the fourth occurrence of the name "Ludwig Van Beethoven" in a <Chapter> element:

```
xpointer(string-range(//Chapter, "Ludwig Van Beethoven")[4])
```

Further Development of XPointer

The need for a fragment identifier technology for XML, and particularly XLink, is undeniable. At the time of writing, there is a lack of available applications that make use of XPointer. This is not surprising, since the XPointer specification continues to remain at Working Draft status and has remained so for an extended period.

Until such time as the XML Linking Working Group can resolve the different views for the most appropriate implementation of XPointer in version 1.0, it is unlikely that application developers will devote substantial time to it. However, once the controversies are resolved, I expect to see a significant number of applications using XPointer, since the functionality promised by the specification is enormously useful.

Further information about XLink, now a full W3C Recommendation, and using XPointer with XLink is found in *XLink Essentials* to be published in 2002.

Looking Ahead

In Chapter 10 we will examine the emerging role of XPath in what I believe will be a particularly important XML technology—XForms.

XForms and XPath

One of the most important new areas in which XPath is beginning to be applied is in the developing XForms specification. This chapter will introduce you to XForms and how XPath is used in it.

XForms is an XML-based technology that is intended to provide forms-based functionality that goes beyond the functionality provided in HTML forms. In particular, XForms will provide a cross-platform means to collect data using a consistent XML-based data model.

In addition, with the explosive growth in the use of XML, it would be a great advantage to submit data from a form to a server in XML format, ready to be processed by XML-capable applications on the server. HTML forms do not send data from forms as XML, but XForms forms do.

NOTE **XForms is currently at Working Draft status at the W3C. Please be aware that details presented in this chapter may be subject to change. The purpose of this chapter is to give you a glimpse of an important emerging use of XPath. Working prototype XForms processors are now becoming available and provide an insight into an upcoming further use—a likely major use—for XPath.**

XForms already has a useful implementation, the X-Smiles browser, which is described later in this chapter. Using the X-Smiles browser, the ideas and code presented

in this chapter can be explored. However, remember that you will need to check for any syntax changes in the XForms specification and which draft of XForms the version of the X-Smiles browser you download supports. The use of XForms, and XPath within it, is very much cutting edge at the time of writing and details of the specification and implementations may not be fully synchronized when you choose to try this out.

Overview of XForms

The XForms specification, sometimes referred to as *Extended Forms* or *XML Forms*, seeks to further develop the useful functionality that has been available through HTML forms. Since collecting data via Web-based forms is an important part of Web interaction, the XForms specification will likely prove to be an important application of XPath.

HTML forms are widely used in the Web and are the primary means for a user to convey information to a server (for example, registration of personal details on Web sites, making choices in online surveys, making online purchases, etc.). From a business point of view, forms are a key way to gather information about customers and to allow them to request information, provide feedback on products and services, and to place orders online.

As HTML forms have been increasingly widely used, and often for business-critical purposes, shortcomings have become apparent. Some of the weaknesses of HTML forms can be ameliorated by using ECMAScript (JavaScript) (to provide client-side validation of data entered on a form, for example), but W3C perceives that to be a partial solution, at best. The W3C XForms specification seeks, using an XML-based XForms model, to address those weaknesses and to provide improved forms functionality. Two important weaknesses identified by the XForms Working Group relate to the need to separate the purpose and presentation of a form and the restricted representation of data captured by an HTML form.

XForms explicitly addresses both areas of perceived weakness. The "purpose" and "presentation" of an XForm are separated. This opens up new ways to express a single data model on multiple platforms. This is something that is, practically speaking, impossible with HTML forms since the structure of the data collected is so closely bound to HTML elements, such as the <input> element, which may not be available on mobile platforms and are inappropriate or irrelevant for voice browsers.

The "data" collected in XForms forms can be handled and validated in ways not available to conventional forms, by making use of the datatyping facilities of XSD Schema (also called XML Schema). We will return to these topics in more detail a little later in this chapter.

Since XForms is explicitly seen by W3C as the replacement for HTML or XHTML forms, a conscious decision was made by W3C to abandon any attempt at full backward compatibility with HTML forms in order to obtain an improved cross-platform solution.

NOTE The version of the XForms specification current at the time of writing is the June 2001 Working Draft. That version is located at www.w3.org/TR/2001/WD-xforms-20010608/. Updates to the XForms specification will be located at www.w3.org/TR/xforms/. If that latter URL shows the June 2001 Working Draft, then no changes have been made to the current version.

In order to give you an impression of what XForms is (or is not), let's make some comparisons between XForms forms and HTML/XHTML forms.

Differences between HTML/XHTML Forms and XForms

There are many differences between XForms and HTML forms. Some of these are listed here.

- Forms has no <form> element as used in HTML.
- The structure and types of form controls has changed, for example, there is now a <textbox> element.
- The XForms form control is not hard wired to a particular way of presentation, so can be opened up for cross-platform visual and/or aural presentation.
- XForms form controls always have captions and hints associated with them, automatically providing users with assistance in filling in a field.
- Data from XForms is submitted as XML.

In HTML, the purpose of a form is not necessarily clear. Typically, because we have seen many forms, we can quickly guess at what it is designed to do. But, the purpose and presentation of HTML forms are tangled together. If it is designed to collect data, quite often that is expressed in the presentation metaphor of the HTML <input> tag and related tags or attributes. When it comes to collecting data across different platforms, it may not be possible to create, for example, an <input> tag with aural CSS or VoiceXML.

Presentation of HTML forms would benefit in some settings from assistance to the user as to how a specific part of a form should be filled in. XForms forms provide that information in two formats. There is a long form (defined behind the scenes in an XForms <caption> element) and a tooltip-type functionality (defined in a <hint> element).

Typically, the content entered into an HTML form's fields are either unchecked for validity or may be checked (partially or completely) using a scripting language, such as JavaScript. XForms makes use of the extensive datatype facilities provided by XSD Schema (also known as W3C XML Schema), therefore providing a high quality of data validation automatically before data is forwarded to the server. Data can be very specifically constrained using enumerations, or by checking its length or the structure of a string of characters entered into a field on an XForms form.

How It Looks to the User

To help you gain a feel for XForms, let's take a brief look at a couple of the demonstration forms that you can access online using the X-Smiles browser.

In Figure 10.1 you can see the on-screen presentation of model.xml, one of the XForms demonstration files available with the X-Smiles browser. To access it online you need to have downloaded and installed the X-Smiles browser. Visit www.xsmiles.org for the download.

This screen shot shows one of the simple potential benefits of XForms. For each field that is to be filled in, a hint as to the content of the field can automatically be supplied to users when they mouse over the field.

XForms forms are designed to be displayed within a variety of other XML application languages. The XForms example in Figure 10.1 is shown in the X-Smiles browser and is displayed using XSL-FO (Extensible Stylesheet Language Formatting Objects).

Conceptually, the screen in Figure 10.1 can be viewed, in the XForms jargon, as purpose, presentation, and data. The purpose of this form, as for many others, is the col-

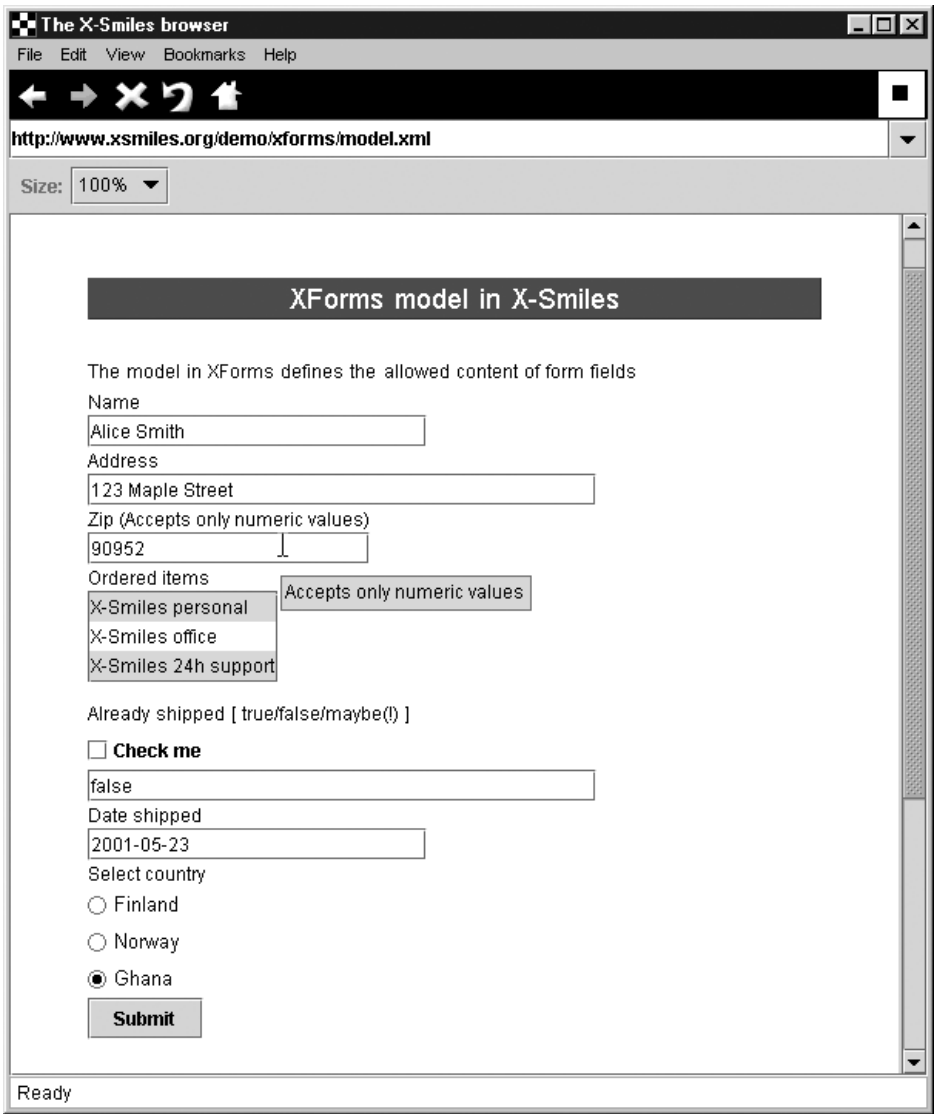


Figure 10.1 A Demonstration XForms Form on the X-Smiles Web Site.

lection of data. In this case, the specific purpose is the collection of shipping data relating to a specific purchase of (fictional) wares offered on the X-Smiles site. The presentation on a desktop browser, such as the X-Smiles browser, is as you see in Figure 10.1. However, many other presentations are possible, for example, on a mobile browser or using aural Cascading Style Sheets. Finally, the data collected in the XForms form can, in a live application, be submitted to a server for further processing when you click on the Submit button.

On the surface, what you see as a user is not very different from an HTML form. This should ease acceptance of XForms by end users. However, what is going on behind the scenes is significantly different from what happens with HTML forms. In this chapter I will introduce you to additional aspects of the structure and operation of XForms, particularly how XPath makes it all work together.

Separating Purpose and Presentation

The value of separating purpose and presentation might not be immediately obvious, so let's look at a simple example to demonstrate how it could be useful.

Let's suppose we wanted to create an order form for the purchase of wotsits. The purpose of collecting the information would remain the same no matter what platform it was being collected on. We would need to know, for example, who was doing the ordering, where the item needed was to be delivered, how the potential purchaser proposed to pay for the item, and, of course, if the purchaser had the bank balance or credit facility to make a valid paid-for purchase. We would want to collect that information whether the XForms form was on a desktop browser, a mobile browser, or a voice browser. The information to be collected is defined in the XForms instance data, which nested within an <xform:xform> element would look something like that in Listing 10.1.

Having decided the XForms model, we have several choices. We could, for example, combine it with another XML application language, such as SVG or XSL-FO; we could use it with XHTML; or, for mobile browsers, we could combine it with a presentation format to be expressed in the Wireless Markup Language (WML).

The creation of the XForms model and definition of the instance data might be done by an expert in XML and database technology, while the presentation in SVG might be done by a graphic designer and the presentation in WML by an expert in that domain. For larger organizations at least, specialist technicians can bring expertise to bear to obtain the optimum benefit from the separation between purpose and presentation that XForms provides.

However, if we used HTML forms for collecting data from desktop browsers, we would have to create a similar model to that for the HTML version, but rewritten so that it is expressed in WML to collect data on mobile platforms. To keep the desktop and mobile browser implementations consistent over time adds to the maintenance task. Clearly, in a multiplatform environment, close ties, as in HTML forms, between the presentation of the material and the information being collected has significant disadvantages.

Let's look at the visible presentation of the form, which is determined by the XForms form controls. It is with form controls that we begin to see how XPath is important in XForms.

```
<xform:xform xmlns:xform="http://www.w3.org/2001/06/xforms">
<xform:submitInfo target="http://www.XMML.com/Purchases/"
method="POST"/>
<xform:model id="" ref="">
<!-- The content of the <xform:model> element is currently undefined. -->
</xform:model>
<xform:instance>
<!-- Notice that the content of the <xform:instance> element is not in
  the XForms namespace. -->
<WotsitOrder xmlns="http:www.XMML.com/Wotsits/">
<Person>
<FirstName></FirstName>
<LastName></LastName>
</Person>
<ShippingAddress>
<Street></Street>
<City></City>
<ZipCode></ZipCode>
<Country></Country>
</ShippingAddress>
<Payment>
<CreditCard>
<CardType></CardType>
<CardNumber></CardNumber>
<CardExpiry></CardExpiry>
</CreditCard>
<Account>
<AccountName></AccountName>
<AccountPassword></AccountPassword>
</Account>
</Payment>
</WotsitOrder>
</xform:instance>
<xform:bind id="" ref="">
<!-- At present the <xform:bind> element has attributes only. -->
</xform:bind>
</xform:xform>
```

Listing 10.1 A Basic XForms Form (XFormExample.xml).

Form Controls

The XForms specification defines several form controls (also referred to as user interface controls) which, for example, allow single selections to be made among a number of defined choices presented in the form. Form controls have been designed with a view to use on many platforms, for example, including compatibility with Aural CSS stylesheets to allow nonvisual presentation of the form.

Form controls are expressed as XForms elements, and their behavior may be modified by XForms attributes and styled using CSS properties. XForms form controls may have their appearance changed by applying CSS stylesheets and can therefore have an appearance similar to typical HTML form controls or may have a very different appearance on screen. Clearly, in an aural browser, the XForms form controls have no visual appearance at all, but they can be presented to the user in some appropriate nonvisual way.

Form controls are bound to the underlying instance data by means of binding expressions contained in the ref attribute of the form control element. The ref attribute, because of its function, is sometimes referred to as a binding attribute. The valid values that can be taken by a binding attribute are always XPath expressions.

A datatype may be bound to a form control. In that situation the datatype restricts the lexical values, which may validly be entered into the form control.

NOTE The names and detailed syntax of form control elements are potentially subject to change. If you are planning to use XPath in an XForms context, be sure to check details against the latest version of the specification at www.w3.org/TR/xforms.

The XForms form control ref attribute may be used on non-XForms form controls. In this case, it must be appropriately namespace declared. The value of the ref attribute must still contain a valid XPath binding expression.

The Textbox Form Control

The textbox form control is designed to allow free-form data entry. The syntax would look something like this:

```
<xform:textbox ref="order/billTo/street" style="width:60; height:15">
 <xform:caption>Street<xform:caption>
 <xform:help>Please enter the number and street name to which the bill
   should be sent.</xform:help>
</xform:textbox>
```

The entered value would be treated as a lexical value. A datatype bound to the form control would be treated as a restriction on the allowed entered value.

Notice that the value of the ref attribute is an XPath expression. In XForms such XPath expressions are called *binding expressions*, since they bind the content of a particular form control to a particular part of the instance data.

All form controls may have element children. The available elements are <caption>, which is required, and <help>, <hint>, and <onevent>, which are optional.

The style attribute may apply style directly to an individual <textbox> element. An alternative approach is for the styling information to be held in an external CSS style sheet, with a particular rule for the <textbox> element.

NOTE Styling from an external CSS style sheet makes use of a class attribute, which can be placed on a <textbox> element or any other form control element.

An implementation may display a <textbox> form control as more than one input area. For example, if the desired information was a date, it might be appropriate to display three input areas with separate input areas for the day, month, and year.

The Secret Form Control

No, I am not telling you something I shouldn't mention. The so-called secret form control is similar to a password field in an HTML form. The value entered into the field is not echoed back in plain text on screen. The code for an XForms secret form control might look like this:

```
<xform:secret ref="XMML/Login/Password">
<xform:caption>Your password</xform:caption>
<xform:help>Enter your password. Be careful that someone isn't looking
  over your shoulder as you do so.
</xform:help>
</xform:secret>
```

Again, the <xform:secret> element has a ref attribute that contains a binding expression which binds the form control to an appropriate part of the instance data.

The uploadMedia Form Control

This form control will provide the functionality to upload media files. The uploadMedia form control will also be able to accept input from a variety of devices, including microphones, digital cameras, and scanners.

Due to the function of the uploadMedia form control, it can be bound only to datatypes xsd:base64Binary or xsd:hexBinary or datatypes that are derived from those.

The selectOne Form Control

This form control allows users to make a single selection from several options. This particular form control could be presented visually in one of several ways, which provide visual appearances similar to those you are familiar with in HTML forms and other applications:

- radioGroup
- checkboxGroup
- pulldown
- listbox
- comboGroup

The choice of presentation is indicated by the value of the selectUI attribute.

The code for a selectOne form control will look something like this:

```
<xform:selectOne ref="Camera/Brand" selectUI="listbox">
  <xform:caption>Famous Camera Brands</xform:caption>
```

```
  <xform:choices>
    <xform:item value="Pentacon">Pentacon</xform:item>
    <xform:item value="Canon">Canon</xform:item>
    <xform:item value="Nikon">Nikon</xform:item>
  </xform:choices>
</xform:selectOne>
```

Selecting one of the offered options will set the value of the value attribute in the underlying data instance at the location Camera/Brand. It is an XPath binding expression.

The selectMany Form Control

The selectMany Form Control will allow the user to make a number of selections greater than one from a range of possible selections offered. The following code snippet will give you an idea of the structures that can be used:

```
<xform:selectMany ref="Pizza/Toppings">
  <xform:caption>XMML's Delicious Pizza Toppings</xform:caption>
  <xform:choices>
    <xform:item value="HamMushroom">Ham and Mushroom</xform:item>
    <xform:item value="Pepperoni">Pepperoni</xform:item>
    <xform:item value="Chocolate">Chocolate</xform:item>
    <xform:item value="SeaFood">Sea Food</xform:item>
  </xform:choices>
</xform:selectMany>
```

The schema for XForms indicates that it is possible to nest <xform:choices> elements within each other. That may allow some sophisticated choices but will pose significant issues of how to clearly present such choices.

As with the selectOne form control, the selectMany form control defines the appearance within the value of a selectUI attribute. In the present draft, a radiobuttonGroup can be used as the visual presentation of a <selectMany> element, but this seems inconsistent with the typical usage of similar radio buttons in existing Web forms.

The selectBoolean Form Control

The selectBoolean form control can be used to allow the user to make choices where there are only two possible answers, such as Yes/No or True/False.

```
<xform:selectBoolean ref="Questionnaire/XPath">
  <xform:caption>Do you love XPath?</xform:caption>
  <xform:help>We need this to determine your sanity!</xform:help>
  <xform:choices>
    <xform:item value="true">Yes</xform:item>
    <xform:item value="false">No</xform:item>
  </xform:choices>
</xform:selectBoolean>
```

The value of the value attribute on the <xform:item> element may only take the values of “true” or “false”. The selectBoolean form control is not suitable for making

choices such as providing information about your gender. That would be better done using the selectOne form control, as follows:

```
<xform:selectOne ref="Person/PersonalInfo/Gender">
  <xform:caption>Your Gender?</xform:caption>
  <xform:choices>
    <xform:item value="Male">Male</xform:item>
    <xform:item value="Female">Female</xform:item>
</xform:choices>
</xform:selectOne>
```

The Range Form Control

The range form control is designed to allow entry of a continuous variable. It might appear like this in an online survey of sentiment regarding politicians:

```
<xform:range ref="/Politicians/Feelings" start="-10.0" end="10.0"
  stepSize="0.5">
  <xform:caption>Positive or negative feelings about
    politicians?</xform:caption>
</xform:range>
```

The XPath binding expression, which is the ref attribute, indicates that the value entered will be set at the location specified in the tree of instance data.

The Button Form Control

This provides similar functionality to a button on an HTML/XHTML form.

The Output Form Control

The output form control provides a means of displaying part of the information in the tree of instance data. For example, the output form control might be used to display the total cost (or running total) of the items being ordered in an online shopping basket. Clearly, it would be inappropriate to allow a user to directly alter such information (although they would have the option to alter the goods that they have ordered).

An output form control might include code like this:

```
The current total for your order is: $<output ref="order/runningTotal"/>
```

The value retrieved from the instance data and displayed would be that defined by the XPath expression `order/runningTotal`.

The Submit Form Control

The submit form control would submit part or all of the data contained in the instance data. The code might look like this:

```
<xform:submit xform="OnlineSurvey">
  <xform:caption>Submit your answers to the survey</xform:caption>
</xform:submit>
```

The Reset Form Control

The reset form control sets the part of the instance data to which it is bound (which may be part or all of the instance data) to the initial values.

The code for the reset form control would look like this:

```
<xform:reset ref="OnlineSurvey/PersonalInfo" xform="OnlineSurvey">
  <caption>Cancel your answers and start again</caption>
</xform:reset>
```

Common Child Elements

For some of the form controls described, I have simply used child elements of the form control element with no further explanation. A brief description of such child elements is given here.

The required <caption> child element provides a label to describe its parent form control. The <caption> element can be provided inline as in the code examples that you saw earlier with the XForms form controls. An alternative approach is to use the xlink:href attribute on the <caption> element to access some external file that contains human-readable caption material. This material could be exploited to provide captions appropriate to the user's locale.

The <help> element is optional and can provide a longer description or helpful comment to the user about how they should respond to the form control.

The <hint> element will provide a short tooltip-type message to the user to assist with responding to the form control.

NOTE The respective roles of the <caption>, <help>, and <hint> elements are unclear in the current Working Draft.

The <onevent> element can be used to bind event handlers to individual form controls.

The <item> element is used within list form controls, such as <selectOne> or <selectMany>, to represent a single item within the list. It may be nested within a <choices> element.

The <choices> element is used within list form controls to group choices that are available to the user.

Moving from HTML to XForms

In this section I will describe a technique for converting XHTML forms to XForms.

If we begin with a very simple form like that in Listing 10.2, we can see in Figure 10.2 (shown in the Internet Explorer 5.5 browser) two types of commonly used form controls: a pair of radio buttons and two text boxes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
          "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<body>
<h3>Online Survey: Personal Details</h3>
<form action="http://example.com/submit" method="post">
  <span>Gender & Age: </span>
  <label><input type="radio"name="gender"value="male"/>Male</label></lable>
    <input type="radio"name="gender"value="female"/>Female</label><br/>
  <label>First Name: <input type="text" name="FirstName"/></label><br/>
  <label>Last Name: <input type="text" name="LastName"/></label><br/>
  <input type="submit"/>
</form>
</body>
</html>
```

Listing 10.2 An HTML Form (HTML.html).

The code shown in Listing 10.2 has no data validation in it, although in practice for HTML forms you may well want to have JavaScript validation being carried out on the client side.

The XForms equivalent would involve nesting code like the following within an <xform> element:

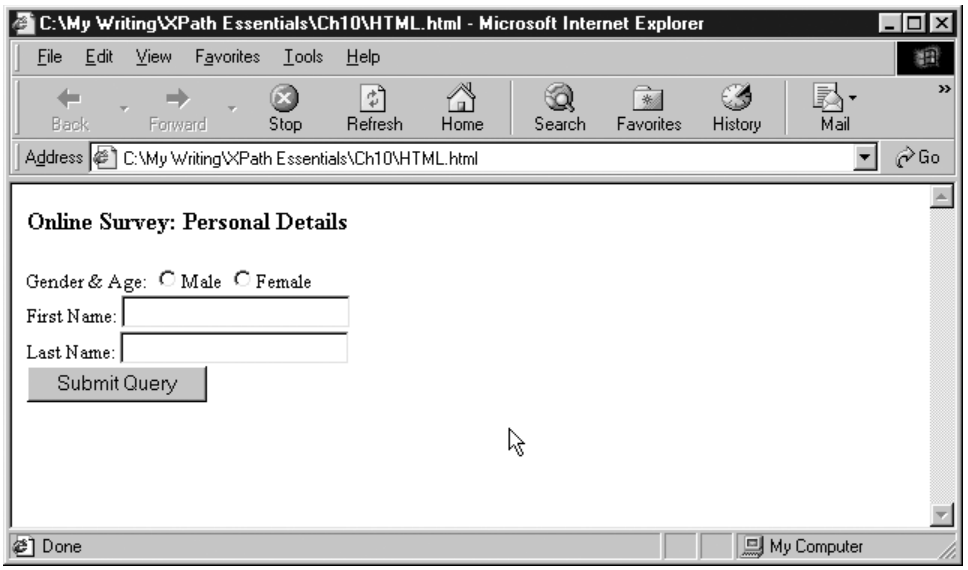


Figure 10.2 HTML Version of a Simple Form.

```
<xform:selectOne xmlns xform="http://www.w3.org/2001/06/xforms"
  ref="Survey/PersDetails/gender" selectUI="radioGroup">
  <xform:caption>Please indicate your gender</xform:caption>
  <xform:choices>
    <xform:item value="male">Male</xform item>
    <xform:item value="female">Female</xform item>
  </xform:choices>
</xform:selectOne>
<xform:textbox xmlns xform="http://www.w3.org/2001/06/xforms"
ref="Survey/PersonalDetails/FirstName">
  <xform:caption>Enter your first name</caption>
</xform:textbox>
<xform:textbox xmlns xform="http://www.w3.org/2001/06/xforms"
ref="Survey/PersonalDetails/LastName">
  <xform:caption>Enter your last name.</xform:caption>
</xform:textbox>
<xform:submit xmlns xform="http://www.w3.org/2001/06/xforms"/>
```

I hope you can see from the code that the <selectOne> element provides a description of what is to be done, but does not tie the desired model to its presentation in the way that a radio button in an HTML form does. The selectUI attribute provides a radio button group, "radioGroup", presentation on screen. Other options available as values for the selectUI attribute are checkboxGroup, pulldown, listbox, and comboGroup.

At the time of writing I am unable to demonstrate the code to you since X-Smiles has not yet been updated to take advantage of the changes from the December 2000 to the June 2001 XForms Working Draft.

Notice that in the ref attribute of the <selectOne> element, we have the following code that looks remarkably like an XPath expression:

```
ref="Survey/PersDetails/gender"
```

That isn't very surprising, since it is an XPath expression. In XForms jargon, that XPath expression is called a *binding expression*, which I will discuss in more detail later in this chapter.

The purpose of the binding expression is to bind the form control, such as the <selectOne> or <textbox> element, to a particular part of the instance data. The instance data is held in memory in a hierarchical structure. That should sound familiar, from the earlier discussion of the XPath data model. The binding expression is a way to navigate to a particular node in that hierarchical tree. If, for example, an entry is made in the <selectOne> element then the node in the tree representation of the instance data has its value altered to correspond with the value(s) entered by the user on the XForms form. Since the binding expression occurs within a ref attribute, the XForms processor will add data entered into (or edited in) the form control to the appropriate part of the instance data held in memory.

Once the form has been completely filled out and the user has hit the Submit button, the data collected on the XForms form will be submitted in XML format. The code sent to the server might look something like this:

```
<?xml version="1.0"?>
<Envelope>
<Body>
<Survey>
<PersonalDetails>
<gender>Male</gender>
<FirstName>Andrew</FirstName>
<LastName>Watt</LastName>
</PersonalDetails>
</Survey>
</Body>
</Envelope>
```

The data sent back to the server corresponds to the binding expressions in the ref attributes of the form controls in our example code.

How XForms Works

In this section I will attempt to convey a little of how an XForms processor works and how XML documents that contain XForms should be written. For a fuller description, consult the XForms specification at the URL given earlier.

XForms can be visualized in a way similar to the relationship between a source document and the XPath in-memory hierarchical representation. In addition there is a presentation of the XForms form, typically visually but also possible as an aural form. I will discuss this as if it were a visual representation.

The source code for an XForms form may typically be embedded in another XML document, although it can be a standalone XForms document. For example, the XForms code might be embedded in the <head> section of an XHTML page.

A tree hierarchy in memory is associated with the XForms form, which represents in a hierarchy the structure of the XForms <xform:instance> element that defines the instance data that the form can collect and typically return to a server. That instance data tree has a root node, and it can be navigated using XPath location paths.

A visual presentation of the form controls corresponding to that same instance data is presented to the user, perhaps including default values contained within the <xform:instance> element. When the user enters or alters a value in the visual presentation, the corresponding node in the instance data tree has its value set accordingly.

XForms User Interface

The XForms user interface is being designed to provide a more functional experience for the user.

User Interface—Dynamic Interface

XForms will provide a dynamic user interface. Let me explain what I mean. Let's suppose you are filling in an online survey and you enter your gender. If you are female you

might be presented with one set of further questions or if you are male you might see a different set, but those supplementary questions are only made visible when you have indicated your gender. Or, if you were making a purchase online and you indicate that you are located in the United States, then XForms can present appropriate further questions about zip codes and present prices in dollars. If you were located in the United Kingdom, it might ask about postal codes and quote prices in pounds sterling. Alternatively, the supplementary questions could be grayed out until some information had been entered in response to the base question.

The <xform:switch> element will allow options like this. For example, we might have the following code:

```
<xform:switch id="location" default="initial">
  <xform:case id="us">
    <xform:group>
      <xform:caption>Please Specify a US zip code.</caption>
    </xform:group>
  </xform:case>
  <xform:case id="uk">
    <xform:group >
      <xform:caption>Please specify a UK postal code.</caption>
    </xform:group>
  </xform:case>
  <xform:case id="initial">
    <xform:group >
      <!-- Some default option. -->
    </xform:group>
  </xform:case>
</xform:switch>
```

The display defined within the <xform:switch> element would be controlled by a <xform:toggle> element, which is an event handler that activates or deactivates parts of the user interface.

```
<xform:toggle switch="switchID" case="caseID" />
```

If we indicated that the country was the United States, the code might look like this:

```
<xform:toggle switch="location" case="us"/>
```

It would then activate a display of the zip code question and any other specific display that was appropriate to a U.S. location.

Such dynamic interfaces can be very useful, although it will be important for implementers of XForms to ensure that the possible sudden change in visual appearance of the form will not cause confusion for the user.

User Interface—Repeating Items

The XForms user interface will also allow items such as line items in an online purchase to be repeated and/or grouped by combining basic building blocks.

The functionality will be provided using <repeat>, <insert>, <delete>, <scroll>, <getRepeatCursor>, and <setRepeatCursor> elements.

User Interface—Interface Templates

Just as it is possible to create new datatypes, so will XForms permit the creation of new user interface components that can be reused or which can be combined or extended to provide complex but functional interfaces. This aspect of XForms is particularly fluid at the time of writing; therefore, I will not describe it further here. If it is of relevance to you, check the latest version of the specification.

User Interface—Layout

The <group> element will allow form controls to be grouped in an appropriate hierarchy. XForms processors will use a box layout mechanism similar to that used in Cascading Style Sheets or XSL Formatting Objects.

It is a design decision yet to be finalized as to whether or not XPath expressions will be able to be used in the context of the hierarchy of layout elements. Check the current version of the specification to see how the conflict between ease of authoring and ease of implementing XForms processors has been resolved.

The XForms Model

An XForms model consists of model items, which incorporate XSD Schema datatype information, as well as properties that are specific to XForms.

NOTE The XForms Model is a part of the XForms Working Draft that is least developed; therefore, this section can only be indicative of what might come in a later version of the specification.

In the following code snippet, referring to a credit card payment, the first line expresses the notion that the credit card number is relevant if the value of the type attribute of the payment element is equal to “credit”.

```
relevant="value('payment/@type') == 'credit'" required="true"
datatype of "xform:string"
facet pattern of "\d{14,16}"
```

In this context, the credit card information is required. It has to be provided in the form of a string and that string must, according to the regular expression in the final line, consist of numeric digits and be between 14 and 16 digits in length.

At present, the details of which possible model items are to be included, or not, in the XForms specification as model items is undefined. However, the properties have been defined in some detail.

Model Item Properties

XForms model items properties can be considered to be in two main categories. The first is computed XPath expressions, which provide a value to the XForms processor, which may be processed on more than one occasion. The second category consists of static values, which the XForms processor evaluates only once.

The name Property

The name property provides a specific name for a model item. It is not a computed expression. Legal values are of the datatype xsd:NCName. There is no default value.

The type Property

The type property assigns a datatype to a model item. It is not a computed expression. Legal values are of the datatype xsd:QName. The default value of the type property is `xsd:anyType`.

The readOnly Property

This describes whether or not the value is restricted from changing. The readOnly property is a computed expression. Legal values are any expression that is convertible to a Boolean value. The default value is "false".

The required Property

The required property defines whether or not a value is required before the instance data is submitted. The required property is a computed expression. Legal values are any expression that is convertible to a Boolean value. The default value is "false".

The relevant Property

The relevant property indicates whether the model item is or is not currently relevant to the XForms model. The relevant property is a computed expression. Legal values are any expression that is convertible to a Boolean value. The default value is "true".

The relevant property helps to control which parts of an XForms form are made visible, or accessible, to a user. For example, if during an online survey, a user indicated that she was not a home owner, then it would not be relevant to display, or allow access to, a series of questions about details of a current loan to purchase her home. The XForms relevant property would, in that situation, be set to "false" by the answer the user had provided, and the display of further questions on that topic would be appropriately constrained.

The calculate Property

The calculate property indicates whether or not the instance data item associated with the model item is to be dynamically calculated. The calculate property is a computed expression. Legal values are an XPath expression convertible to an XPath datatype compatible with the associated XSD Schema datatype. There is no default value.

The priority Property

The priority property indicates the relative priority for calculation of the model item. It is not a computed expression. Legal values are any expression that is convertible to an integer in the range 0 to 32767. The default value is 0.

The validate Property

The validate property specifies the predicate that needs to be satisfied for the associated data instance item to be considered valid. The validate property is a computed expression. Legal values are any expression that is convertible to a Boolean value. The default value is "true".

Using Datatypes in the XForms Model

In the Working Draft current at the time of writing, it is clear that the use of datatypes in XForms will be modeled on the XSD Schema. However, substantial uncertainty remains about the detail of how this will be done.

Let's suppose that a U.S. state is to be entered as part of an address. When using XSD Schema, that would be declared as being of datatype xsd:string, but with the standard abbreviations for U.S. states the string entered must be two characters in length:

```
<xsd:simpleType>
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="2"/>
    <xsd:maxLength value="2"/>
  </xsd:restriction>
</xsd:simpleType>
```

or more succinctly as

```
<xsd:simpleType>
  <xsd:restriction base="xsd:string">
    <xsd:length value="2"/>
  </xsd:restriction>
</xsd:simpleType>
```

Thus any attempt to enter a string of fewer than two characters in length or more than two characters in length would cause the XForms processor to indicate that an unacceptable value had been entered.

Similarly, when making an online payment to a store, only certain types of credit cards might be accepted. This scenario could be represented by an enumeration restriction, something like this:

```
<xsd:simpleType>
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Mastercard"/>
    <xsd:enumeration value="Visa"/>
    <xsd:enumeration value="American Express"/>
```

TEAMFLY

```
    </xsd:restriction>
  </xsd:simpleType>
```

Alternatively, a merchant may have preferred credit cards but will leave open an additional option for the purchaser to enter another type of card. This is called an open enumeration and can be expressed like this:

```
<xsd:simpleType>
  <xsd:union memberTypes="xsd:string">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Mastercard"/>
        <xsd:enumeration value="Diner's Club"/>
        <xsl:enumeration value="Visa"/>
        <xsd:enumeration value="American Express"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Some XForms form controls, such as <selectMany>, have the idea of choosing more than one simpleType item at a time. This corresponds to XSD Schema list datatypes. Thus, a list that offered a choice of more than one string could be expressed in XSD Schema like this:

```
<xsd:simpleType name="listOfMyStringType">
  <xsd:list itemType="xsd:string"/>
</xsd:simpleType>
```

Full discussion of the use of XSD Schema datatypes is beyond the scope of this chapter, but I hope these examples have given you a flavor of how things are likely to work in XForms. Further information on XSD Schema can be found in Wiley's XML Schema Essentials.

XForms Terminology

XForms introduces a data model that is distinctly different from that in HTML/XHTML forms, although its details have not yet been fully defined. As a result, XForms also introduces quite a few new terms that you will need to understand if you are to grasp precisely how XForms works. This section provides a mini-glossary for XForms, which may help you understand how it works. Note that several terms are drawn from the XSD Schema Part 2 (Datatypes) Recommendation (the URL is given below).

- **Binding.** The connection between a form control and a model item and an instance data item, represented as a binding expression.
- **Binding element.** An element that is explicitly allowed to have an xform:ref attribute and therefore uses an XPath binding expression to bind to, for example, an instance data item.

- **Binding expressions.** XPath expressions used to associate a form control with the tree representation of instance data.
- **Computed expression.** An XPath expression used with model item properties (such as `relevant` and `calculate`) to include dynamic functionality in XForms.
- **Containing document.** The document from a non-XForms namespace in which the XForms Data Model is embedded.
- **Datatype.** A 3-tuple, which consists of (a) a set of distinct values, called its value space; (b) a set of lexical representations, called its lexical space; and (c) a set of facets that characterize properties of the value space, individual values, or lexical items. See Part 2 of the XSD Schema Recommendation located at www.w3.org/TR/xmlschema-2/.
- **Facet.** A single defining aspect of a value space. See Part 2 of the XSD Schema Recommendation for further information.
- **Form control.** A user interface control which acts as a focus for user interaction.
- **Immediately enclosing element.** The first binding element node in the node-set returned by the XPath expression `ancestor::*`.
- **Instance data.** An in-memory tree representation of the values and state of all the instance data items associated with a particular form.
- **Instance data item.** An in-memory representation of the value and state of a single piece of data corresponding to an XSD Schema simpleType, constrained by the definition of a model item.
- **Lexical space.** A lexical space is the set of valid literals for a datatype. See XSD Schema Part 2 for further information.
- **Model item.** An abstract unit of data collection that typically defines an XSD Schema datatype.
- **Model item property.** A single XForms-specific defining aspect of a model item.
- **Value space.** A set of values for a given property. Each value in the value space of a datatype is denoted by one or more literals in its lexical space. For further information, see the XSD Schema Part 2 Recommendation.
- **XForms model.** The nonvisible definition of an XML form as specified by XForms. The XForms model defines the individual model items and constraints and other run-time aspects of XForms.

XForms Elements

XForms provides several elements that form the structural backbone of an XForms form.

The xform Element

The <xform:xform> element is the element root of a freestanding XForms form or, when an XForms form is nested within a document from another XML namespace, is the element within which all XForms elements are nested.

The model Element

The <xform:model> element is used to link the XForms form to a Document Type Definition or an XSD schema. For example, if we wanted to associate a credit card payment XForms form to a particular schema we could use code like this:

```
<xform:model href="XMMLccpayments.xsd">
</xform:model>
```

Alternatively, the XForms model can be defined inline.

The instance Element

The <xform:instance> element can be used to define initial values for instance data. The instance data may be accessed from a URI or may be defined inline. The content of the <xform:instance> element may be in any namespace other than the XForms namespace.

The submitInfo Element

The <xform:submitInfo> element defines how and where submitted information is to be sent. For example, if we wished to submit information using the HTTP POST method, our code would look like this:

```
<xform:submitInfo
  xlink:href="http://www.XMML.com/SurveyResults/"
method="POST">
<!-- Content goes here. -->
</xform:submitInfo>
```

The bind Element

The <xform:bind> element binds different parts of an XForms form together. For example, to refer to a particular external source of an XForms model, we would use code like this:

```
<xform:bind
  ref="http://www.XMML.com/MyXFormsDataModel.xml">
</xform:bind>
```

XForms Properties

For each XForms element, an XForms processor is expected to maintain a set of read-write properties, which are described in this section.

The immediate-refresh Property

The immediate-refresh property controls whether or not any change to instance data is immediately reflected in the data displayed in the user interface.

The immediate-revalidate Property

The immediate-validate property controls whether or not any change to instance data causes a validation of the data to be carried out immediately.

The immediate-recalculate Property

The immediate-recalculate property controls whether or not a change to particular parts of the instance data causes a recalculation to be carried out. For example, if the number of items to be purchased is altered, the immediate-recalculate property controls whether or not the total for the cost of an order is immediately recalculated.

The use-nils Property

The use-nils property controls whether or not nils, from XSD Schema Instance, are or are not used in instance data.

The Read-Only Properties

In addition to the properties just described, XForms also provides access to a number of properties that are read-only.

- The version property describes the version of the XForms specification being used. For XForms 1.0 it is a string with the value of "1.0".

The conformance-level property indicates the level of conformance with the XForms specification that a particular XForms processor aims to achieve. At present it is envisaged that two levels of conformance will be indicated by the property. It is likely that hand-held devices, Web-enabled mobile phones, etc., will implement XForms Basic. Desktop clients and servers will likely implement XForms Full.

The timezone property provides information about the offset of the local time zone from Greenwich Mean Time. The offset is expressed in minutes.

XForms Processors

It is currently too early for definitive XForms implementations to be available. Fortunately for those who are interested in the topic, the X-Smiles multi-namespace XML browser provides a very interesting and useful test bed to explore XForms and how it applies XPath in a new context.

The X-Smiles Browser

The X-Smiles browser is a multi-namespace browser that can process XForms as well as other XML application languages such as Scalable Vector Graphics (SVG), the Synchronized Multimedia Integration Language (SMIL), and the XSL Formatting Objects (XSL-FO).

X-Smiles is a Java-based browser that is still a prototype product—at the time of writing it is version 0.33. Further information about the X-Smiles browser is available at www.xsmiles.org. A free download is available to allow you to explore the use of XPath in XForms, as well as possibly trying out a number of the other new XML application languages mentioned earlier. Since X-Smiles is under ongoing development, you will need to consider backing up all important files, if you choose to download and run it. However, my own experience with it has been that it has run smoothly, causing no side effects on other programs or data.

NOTE **If you have unexpected difficulty downloading the X-Smiles browser and are attempting to do so from a corporate base, consider the possibility that someone in your IT department has put a bar on your downloading from an "X-rated" adult site! This has happened.**

Mozquito XForms Preview Release

A preview release of an XForms processor can be downloaded from the Mozquito.com Web site.

Cardiff.com LiquidOffice

Cardiff.com offers an evaluation version of their LiquidOffice product, which incorporates functionality from the XForms Working Draft. Check their Web site for the latest information.

XPath in XForms

XPath is used extensively throughout XForms. The ref attribute of form controls reference the in-memory version of instance data using binding expressions. Additionally, the XPath id ( ) function may be used to access the representation of elements that possess an ID attribute.

Instance Data

Often an XForms author will want to ensure that the data entered in the form corresponds to a particular structure. XForms provides more than one way to achieve this.

One option is to validate the data entered against a Document Type Definition or an XSD schema. The association with the XSD schema may be expressed like this:

```
<xform:model href="MySurvey.xsd">
</xform:model>
```

Another option is to provide instance data to the XForms processor embedded in the form. Initial instance data is provided in an <instance> element nested within the

<xform> element. The <instance> element must follow the <submitInfo> element in the sequence of child elements of the <xform> element.

For the online survey that we described earlier, that might look like this:

```
<xform:instance xmlns="http://www.XMML.com/Survey/">
<Survey>
<PersonalDetails>
<gender>Male</gender>
<FirstName></FirstName>
<LastName></LastName>
</PersonalDetails>
</Survey>
</xform:instance>
```

The content of individual elements that make up the instance data may be empty or, in a way similar to HTML forms, a default value may be indicated. For example, if we wished to make the value "Male", the default option for the <gender> element, we could indicate that as in the above code.

The instance data will typically be in its own namespace. There is within the start tag of the <instance> element a namespace declaration that will be used when the XML, as modified by user input, is uploaded to the server. While it is held in the XForms processor, the instance data will be represented as an in-memory tree hierarchy and individual nodes within the instance data can be accessed using XPath binding expressions. When the instance data is submitted to the server, it is serialized before being sent.

The location to which the serialized instance data is submitted is indicated by the target attribute on the <submitInfo> element, like this:

```
<xform:submitInfo target="http://www.xmml.com/onlinesurvey" method="..."/>
```

If the person filling in the survey is female, then when the appropriate form control is completed, the value in the in-memory instance data will be updated to reflect the user-initiated change. The form control will be linked to the in-memory hierarchy by use of the binding expression in the ref attribute of the form control element.

XPath Context in XForms

You may recall that the formal definition in XPath of the context is fairly complex. The equivalent definition in XForms is also less than immediately obvious. I will divide the definition of the context into that for outermost binding elements and non-outermost binding elements.

Context for Outermost Binding Elements

The context for outermost binding elements has six parts:

- A context node. In this case, the root node.
- A context size, exactly 1.

- A context position, exactly 1.
- No variable bindings.
- An available function library as defined in the XForms specification.
- Any namespace declarations in scope for the attribute that defines the expression are applied to the expression.

Context for Non-outermost Binding Elements

The context for binding elements other than the outermost is similar to that shown earlier.

- A context node, defined by evaluating the binding expression of the immediately enclosing element.
- A context size, exactly 1.
- A context position, exactly 1.
- No variable bindings.
- An available function library as defined in the XForms specification.
- Any namespace declarations in scope for the attribute that defines the expression are applied to the expression.

Binding Expressions

XForms' binding expressions use a syntax that very closely resembles XPath abbreviated syntax. A binding expression is typically used in the value of the ref attribute of a form control, thereby binding any data input into that part of the form control to a particular part of the instance data for the form.

If we had simple credit card payment details held in instance data structured like the following, we could create an XForms form that reflects that structure within its user interface controls.

```
<Payment>
<NameOnCard></NameOnCard>
<Address country=""></Country>
<CardNumber></CardNumber>
<ExpiryDate></ExpiryDate>
<Currency></Currency>
<Amount></Amount>
</Payment>
```

Note that there is a binding expression in the ref attribute of form controls. Each binding expression is similar to those you have seen in earlier parts of the book used to select elements and attributes for processing by an XSLT processor. In this context they are selecting elements and attributes for the purpose of informing an XForms processor

of where data entered by a user is to be inserted or updated in the in-memory hierarchy of nodes.

Canonical Binding Expressions

As you saw in Chapter 4, there can be several ways of expressing the same XPath expression or location path. If we wanted to select the <Chapter> element in this simple document

```
<Book>
<Chapter>
</Chapter>
</Book>
```

we could use the following in the unabbreviated absolute syntax

```
/child::Book/child::Chapter
```

and

```
/Book/Chapter
```

in the abbreviated absolute syntax. The two expressions mean exactly the same thing.

In addition, if this was expressed, say, in the select attribute of an <xsl:apply-templates> element in the main template, we could also write

```
child::Book/child::Chapter
```

in unabbreviated relative syntax and

```
Book/Chapter
```

in abbreviated relative syntax and again have the same logical meaning in that the same node set (of one node) would be selected.

Just as with the Canonical XML 1.0 Recommendation (www.w3.org/TR/2001/REC-xml-c14n-20010315), there are advantages in having one unambiguous canonical form to represent all the logically equivalent possible forms. In XForms the canonical form of XPath is the abbreviated absolute syntax.

Thus, in XForms, the canonical binding expression uses the abbreviated absolute syntax.

Datatypes

As you will recall, XPath 1.0 uses only four datatypes: node set, Boolean, string, and number. XForms can use the four XPath datatypes, but is also capable of using the full gamut of XSD Schema (also called W3C XML Schema) datatypes. An XForms processor is able to distinguish the group of datatypes that any individual datatype is associated with, according to the context in which a datatype is being used.

XForms binding expressions and computed expressions use the XPath datatypes. All other parts of XForms use XSD Schema datatypes.

NOTE **XPath 2.0 is likely to provide full support for XSD Schema datatypes. XForms is expected to adopt XPath 2.0 in a version following XForms 1.0.**

XForms makes extensive use of XSD Schema datatypes, including the XSD Schema notions of value space and lexical space. XForms also includes XSD Schema constraining facets.

Table 10.1 lists the 19 built-in primitive datatypes, defined in XSD Schema Part 2, which XForms uses.

In addition to the built-in primitive datatypes from XSD Schema, XForms also uses the 23 built-in derived datatypes defined in XSD Schema part 2. These are listed in Table 10.2.

The definitions for each of these datatypes, together with full details of their value space, lexical space, and facets are described in Part 2 of the XSD Schema Recommendation.

XForms Masks and Regular Expressions

In XSD Schema there exists a regular expression language, which is similar to the regular expression language in PERL. The XForms Working Group currently believes that

Table 10.1 The Built-in Primitive Datatypes in XSD Schema

duration	dateTime	time	date
gYearMonth	gYear	gMonthDay	gDay
gMonth	string	boolean	base64Binary
hexBinary	float	decimal	double
anyURI	QName	NOTATION	

Table 10.2 The Built-in Derived Datatypes in XSD Schema

normalizedString	token	language	Name
NCName	ID	IDREF	IDREFS
ENTITY	ENTITIES	integer	nonPositiveInteger
negativeInteger	long	int	short
byte	nonNegativeInteger	unsignedLong	unsignedInt
unsignedShort	unsignedByte	positiveInteger	

full implementation of regular expressions may be too complex. The Working Group suggests that the concept of "mask facet," adopted from Wireless Markup Language (WML) 1.3, be used instead.

All XForms mask facets are said to be convertible into regular expressions patterns.

Regular expressions permit more precise control over the structure of a string entered on a form than simple datatyping can achieve. For example, suppose you wanted to have a field for U.S. telephone numbers. They might take the form of

```
(123)4567890
```

That is, an opening parenthesis, three digits, a closing parenthesis, and seven digits. This could be expressed in an XForms mask in the following way:

```
\(NNN\)NNNNNNN
```

meaning that the literal opening parenthesis character is output followed by three numerical digits, followed by a closing parenthesis character, followed by seven numerical digits.

XForms-specific Datatypes

As well as using the XSD Schema datatypes, XForms derives further datatypes that are particularly relevant to use in forms.

The Currency Datatype

The XForms currency datatype is derived from the xform:string datatype. The specified pattern facet is [A-Z]{3}, which means that three uppercase characters are the permitted structure. Thus JPY could represent Japanese yen and USD could represent U.S. dollars.

The Monetary Datatype

The equivalent of a monetary datatype could be created by using an xform:decimal and an xform:currency datatype separately. However, consideration is being given to providing a combined monetary datatype, values of which would take a form of 15.50EUR, meaning 15.50 euros.

Multiple Forms in Containing Document

The XForms specification permits the existence of more than one XForms form in a containing document. When this occurs, there are issues of being able to reliably associate a form control through a binding expression with the corresponding part of the in-memory instance data.

The solution to this issue is for <xform> elements to have an ID attribute. The first <xform> element in document order does not need to have an ID attribute. In the absence of an ID reference, the XForms processor assumes that it is the <xform> element, which is first in document order that is being referred to.

Thus if, in addition to the survey we used in earlier examples, we wanted to include some service for which payment was necessary, we might add another <xform> element to the containing document like this:

```
<xform:xform id="payment">
<xform:submitInfo target="http://www.XMML.com/payment"
 method="..."/>
</xform:xform>
```

The ID attribute allows the second <xform> element to be referenced unambiguously. The content of the <xform> element would be referenced using code like this:

```
<xform:selectOne xform="payment" ref="payment/credit/cardnumber">
```

XForms Function Library

XForms uses all of the XPath 1.0 Core Function Library, which was described in Chapter 2, and discussed in more detail in Chapter 5. XForms does not describe node set functions additional to those in XPath 1.0, but it does provide a few additional number, string, and Boolean functions to complement those provided by XPath.

Number Functions

XForms provides four additional functions that return numbers.

The average( ) Function

The `average()` function returns the arithmetic average value, for each node in the argument node set, of the result of converting the string-values of each node to a number.

The min( ) Function

The `min()` function returns the minimum value, for each node in the argument node set, of the result of converting the string-values of the node to a number.

The max( ) Function

The `max()` function returns the maximum value, for each node in the argument node set, of the result of converting the string-values of the node to a number.

The count-non-empty( ) Function

The `count-non-empty()` function returns the number of non-empty nodes in the argument node set. A node is considered non-empty if it is convertible into a string with a greater-than-zero length.

The count-non-empty ( ) function is useful when determining whether or not all fields in a form have been filled in.

String Functions

XForms adds two new string functions to those available in the XPath 1.0 Core Function Library.

The now( ) Function

The now ( ) function returns the current system time as a string value. If local time zone information is available to the XForms processor, that is also returned in the string.

The xforms-property( ) Function

The xforms-property ( ) function takes the name of an XForms property as a string argument, accesses that property, and returns its value as a string.

Boolean Functions

In conventional HTML forms, the submit and reset functionality is a basic provision. In XForms equivalent functionality is provided by the new submit() and reset() functions that return Boolean values.

The submit( ) Function

The `submit()` function immediately submits the instance data bound to the node that contains the expression by triggering an `xforms-submit` event.

The reset( ) Function

The `reset()` function immediately resets the instance data bound to the node that contains the expression by triggering an `xforms-reset` event.

Looking Ahead

This chapter provided an introduction to XForms, which I expect to be a major use of XPath, given that forms are such a fundamental part of interactivity with the users of all but the simplest Web site.

In Chapter 11 we will examine two new XML-based technologies that are concerned with the security of XML data and that make use of XPath—Canonical XML and XML Signatures.

XPath in Canonical XML and XML Signatures

One of the dichotomies about XML is that it makes the sharing of business information arguably easier than it has ever been, but the other side of the coin is that at least some of that business information is commercially sensitive or confidential and needs to be shared only in a controlled way. Canonical XML and XML Signatures are two of a growing number of XML security specifications being developed at W3C (and elsewhere) to provide security for XML-based information.

Both Canonical XML and XML Signatures make use of XPath and its data model in their functionality. To understand where XPath fits in and what its role is in these XML security technologies, first let's take a quick look at what the W3C is doing so that we can understand where Canonical XML and XML Signatures fit into the W3C strategy. Well then take a closer look at Canonical XML and XML Signatures and how XPath is used with them.

Overview of XML Security Specifications

The W3C has several security-oriented specifications complete or under development. Together, as the pieces of the jigsaw are completed, these will provide a useful basis for a security suite written in XML, which can assist in the provision of an appropriate level of security for XML data and data held in other formats.

The current W3C XML security specifications are

- Canonical XML (www.w3.org/TR/xml-c14n)
- XML Signature Syntax and Processing (www.w3.org/TR/xmldsig-core)
- XML Encryption Syntax and Processing (www.w3.org/TR/xmlenc-core)
- XML Key Management Specification (www.w3.org/TR/xkms)

Canonical XML is a full W3C Recommendation. XML Signature Syntax and Processing is a W3C Proposed Recommendation, and XML Encryption Syntax and Processing is a W3C Working Draft. The XML Key Management Specification is currently a W3C Note.

NOTE The XML Signature Syntax and Processing specification is a joint effort of W3C and the Internet Engineering Task Force (IETF).

A canonical form is a unique, unambiguous physical representation of a set of XML documents that are logically equivalent, as permitted through the syntax alternatives allowed by the XML 1.0 Recommendation. The need for a canonical form, and precisely what a canonical form is, will be explained a little later in the chapter.

XML Signature syntax and XML Encryption syntax act on the canonical form of an XML document. The XML Key Management specification proposes a method of distributing and registering public keys suitable for use with XML Signature syntax and XML Encryption syntax.

NOTE It is not only W3C who are currently developing XML-based security standards. The Organization for Advancement of Structured Information Standards (OASIS) is developing two such specifications: the Security Assertion Markup Language (SAML) and the Extensible Access Control Markup Language (XACML).

Further information on SAML is located at www.oasis-open.org/committees/security/. Additional information on XACML is located at www.oasis-open.org/committees/xacml/.

The initiatives at W3C and OASIS are complementary to a significant degree. The OASIS Technical Committees have indicated their awareness of the W3C initiatives. The likelihood is that with the efforts of W3C, IETF, and OASIS, a useful and likely powerful set of XML security specifications will be made available.

Principles of Security

Computer security is a huge and complex field. In this section I can only hope to give you a brief overview of some main principles. These will help you set Canonical XML and XML Signatures in context.

Important principles of computer security, which in practice are often used in a variety of combinations, include:

- Authentication
- Authorization

- Access control
- Confidentiality
- Data integrity
- Nonrepudiation

Authentication is the process of establishing who a person (or another computer) is, by establishing whether or not the claimed identity is genuine. Typically, authentication may take place by use of a shared secret, such as a password. Authentication is important because you want to know who is requesting information, who is attempting to conduct a transaction, who was the author of a document, etc. The XML Signatures specification is particularly relevant to authentication.

Typically authentication will not be used in isolation. The reason for establishing who a person is or which computer, data, or services he or she is attempting to gain access to is to provide a means for appropriately granting a user access to information, according to predefined business rules. In this setting, authentication will be combined with appropriate authorization.

Authorization is the process of assigning or refusing access rights to particular data, information, or services, having assumed or established the identity of a person or a client computer. Authorization may be granted for access to nonsensitive information in the absence of authentication. For example, access to at least parts of most Web sites or to many FTP sites (or parts of them) is commonly allowed without a user having to log in.

Access control is the process of enforcing authorization. It depends on authentication, since the identity of a person or client computer needs to be established before the granted access rights for that individual or computer can be determined. Once the identity and access rights of the user are known, access control ought to reliably allow access to designated areas and reliably prevent access to areas for which authorization has not been granted to that individual. The Extensible Access Control Markup Language (XACML) is particularly relevant for this aspect of security and also has significant relevance for the related topic of confidentiality.

Confidentiality is the process used to protect confidential data from unauthorized disclosure. Confidentiality may use encryption or access control, or both, to protect the content of the files from being read by those without authority. The XML Encryption specification is particularly relevant to confidentiality.

Data integrity is the process of preventing unauthorized or undisclosed alteration of data. When data is sent electronically it is important, particularly for the recipient, to be sure that the document that arrives is the document that the sender actually sent, and that it has not been changed in some way by what is termed a man-in-the-middle attack. The XML Signatures specification is particularly relevant to the issue of data integrity.

Nonrepudiation (also called *nonrepudiability*) is the process of establishing beyond doubt that, for example, a document was created by a certain individual. For example, if you appear to buy 100 books on Amazon.com it is important, from the point of view of the seller, that such a significant purchase be a genuine one. The XML Signatures specification is relevant to the issue of nonrepudiation.

Having taken a very brief look at an overview of computer security, let's turn to the detail of Canonical XML so that we can understand where XPath fits in.

Canonical XML

So, just what is canonical XML and why do we need it? Canonical XML is a way to express an XML document, or members of a set of XML documents, in a way that removes ambiguity created by the flexibility in XML syntax permitted in the XML 1.0 Recommendation. This is done by creating a physical XML document following a precise processing algorithm which, step-by-step, deals with each of the potential ambiguities introduced by the allowed flexibility in XML 1.0 syntax.

To be more precise, canonical XML can also express document subsets in a canonical form. XPath is important in the process of defining a subset of an XML document from which a canonical form can then be constructed.

Why Canonical XML Is Needed

First let's look at the problem that the permitted flexibility in XML syntax brings about.

Any XML document is part of a set of XML documents that are logically equivalent, but which are physically represented in a variety of ways.

That sounds pretty abstract so let's look at what that actually means.

The term *canon* originally referred to the Bible, so we will use that for our example. Look at the XML document in Listing 11.1.

Compare that with the very similar document in Listing 11.2 and see if you can spot the differences.

```
<?xml version='1.0'?>
<Canon version="1603">
<OldTestament language="English">
<Book order="first" chapters="50">
Genesis
</Book>
<Book order="second" chapters="40">
Exodus
</Book>
<Book order="third" chapters="27">
Leviticus
</Book>
</OldTestament>
<NewTestament>
<Book order="first" chapters="28">
Matthew
</Book>
<Book order="second" chapters="16">
Mark
</Book>
```

Listing 11.1 An Abbreviated Listing of Books of the Bible in XML (Canon.xml).

```
<Book order="third" chapters="24">
Luke
</Book>
</NewTestament>
</Canon>
```

Listing 11.1 (*Continued*)

```
<?xml version='1.0'?>
<Canon version='1603'>
<OldTestament language='English'>
<Book  chapters='50' order='first'>
Genesis
</Book>
<Book chapters='40' order='second' >
Exodus
</Book>
<Book order='third' chapters='27'>
Leviticus
</Book>
</OldTestament>
<NewTestament>
<Book order='first' chapters='28'>
Matthew
</Book>
<Book order='second' chapters='16'>
Mark
</Book>
<Book order='third' chapters='24'>
Luke
</Book>
</NewTestament>
</Canon>
```

Listing 11.2 An XML Document That Is Logically Equivalent to Listing 11.1 (Canon02.xml).

The differences are slight. The delimiters of the values of each of the attributes in the first document are double quotes (that is, the " character). In the second document, the delimiters are the apostrophe (that is, the ' character). XML 1.0 syntax treats those two forms of delimiters as being equivalent, but if the documents are being compared character for character, they are not identical.

A second difference is that the order of the attributes in the first two <Book> elements is different in the first version compared to the second. In the first version the order attribute precedes the chapters attribute; whereas in the second, the chapters

attribute comes before the order attribute. Again, if you are familiar with XML 1.0 syntax, you will know that those two are functionally equivalent since XML 1.0 syntax does not expect the attributes of any XML element to occur in any particular order. Thus, to an XML-aware processor these documents are equivalent. However, to encryption software, which has no knowledge of the logical meaning of the physical structure of an XML document, the two documents are significantly different.

Similarly, an XML document may have parsed entities, which define part of its content. Compare Listing 11.3, which has an internal parsed entity, with Listing 11.4, where there is no entity.

If we view either of the documents in Internet Explorer, we see exactly the same document structure shown in Figure 11.1.

If, instead of using an internal entity, we used an external general parsed entity for the Author entity (Listing 11.5) with the content in the file ExternalEntity.txt (Listing 11.6), we have yet another way of expressing the same logical XML structure, this time involving two files to express the same meaning. To a fully XML-aware processor, the external general parsed entity would be used, as replacement text, to replace the entity in the source XML document.

NOTE **The common Web browsers do not presently render external general parsed entities. In Internet Explorer 5.5, for example, the <Author> element will be shown as an empty element on screen.**

```
<?xml version='1.0'?>
<!DOCTYPE Book
[<!ENTITY Author "Andrew Watt" >
]>
<Book>
<Title>XPath Essentials</Title>
<Author>&Author;</Author>
<Publisher>John Wiley</Publisher>
</Book>
```

Listing 11.3 An XML Document Using an Internal Parsed Entity (Book.xml).

```
<?xml version='1.0'?>
<!DOCTYPE Book SYSTEM "Book3.dtd">
<Book>
<Title>XPath Essentials</Title>
<Author>Andrew Watt</Author>
<Publisher>John Wiley</Publisher>
</Book>
```

Listing 11.4 An XML Document with No Internal Parsed Entity with a DTD (Book02.xml).

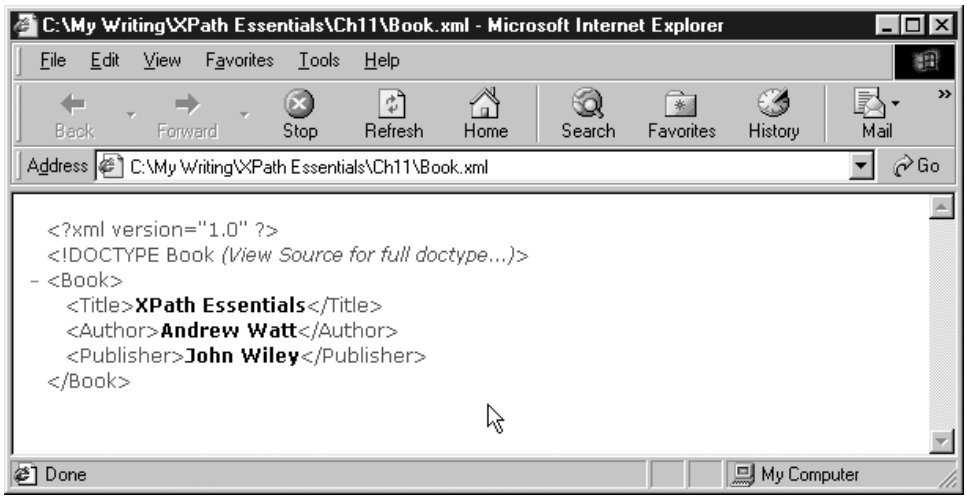


Figure 11.1 The Appearance of the Document with or without the Use of an Internal Parsed Entity.

```
<?xml version='1.0'?>
<!DOCTYPE Book SYSTEM "Book3.dtd"
[<!ENTITY Author SYSTEM "ExternalEntity.txt" >
]>
<Book>
<Title>XPath Essentials</Title>
<Author>&Author;</Author>
<Publisher>John Wiley</Publisher>
</Book>
```

Listing 11.5 An XML Document Using an External Parsed Entity (Book03.xml).

```
Andrew Watt
```

Listing 11.6 A Brief External Parsed Entity, Used by Listing 11.5 (ExternalEntity.txt).

Compare Listing 11.7 with Listing 11.8 and see if you can spot the difference. I don't expect you to, but without Canonical XML these two files would seem different to a processor that was applying, for example, an XML signature to the source document.

The only difference between the two documents is that in the first version there are two space characters between the number attribute name/value pair and the status attribute name/value pair in the start tag of the second <Chapter> element. In the second version there is only one space character there.

Again, this trivial difference in the physical document structure, between two documents that are logically identical, will result in two different results if each document is encrypted, because encryption operates on characters not logical XML tokens.

```
<?xml version='1.0'?>
<Book>
<Introduction status="draft">This is a draft introduction</Introduction>
<Chapter number="1" status="draft">Only a draft so far
</Chapter>
<Chapter number="2"  status="draft">A simple draft for Chapter 2
</Chapter>
</Book>
```

Listing 11.7 An XML Description of a Book (Book04.xml).

```
<?xml version='1.0'?>
<Book>
<Introduction status="draft">This is a draft introduction</Introduction>
<Chapter number="1" status="draft">Only a draft so far
</Chapter>
<Chapter number="2" status="draft">A simple draft for Chapter 2
</Chapter>
</Book>
```

Listing 11.8 An XML Document Logically Equivalent to Listing 11.7, but with a Single Syntax Difference (Book05.xml).

TEAMFLY

Another XML file might contain the following element:

```
<ThisIsEmpty/>
```

Another file conforming to the same schema could, entirely legally, have the same element expressed as

```
<ThisIsEmpty></ThisIsEmpty>
```

Again we have two snippets of XML syntax that are logically equivalent but whose physical representations in a serialized XML document differ. Encryption of two XML documents that are otherwise logically identical will result in two different encrypted files.

Why am I showing you so many such trivial differences between files? The answer is that if any of these groups of logically equivalent but physically (slightly) different files were the input for encryption or a digital signature, they would each produce a different encrypted result.

Similar issues arise if we use different namespace prefixes but associate them with one namespace URI. Listing 11.9 is, as far as a namespace-aware XML parser is concerned, the same as Listing 11.10, since an XML parser uses the expanded name for an element which, as you may remember, consists of the namespace URI and the local part.

```
<?xml version='1.0'?>
<xmml:Book xmlns:xmml="http://www.xmml.com/Schemas">
<xmml:Chapter>Chapter 1 is about XML</xmml:Chapter>
<xmml:Chapter>Chapter 2 introduces XPath</xmml:Chapter>
</xmml:Book>
```

Listing 11.9 A Simple XML Description of a Book (XMMLBook.xml).

```
<?xml version='1.0'?>
<My:Book xmlns:My="http://www.My.com/Schemas">
<My:Chapter>Chapter 1 is about XML</My:Chapter>
<My:Chapter>Chapter 2 introduces XPath</My:Chapter>
</My:Book>
```

Listing 11.10 An XML Document Logically Equivalent to Listing 11.9, but Using a Different Namespace Prefix (MyBook.xml).

The fact that different namespace prefixes are used is essentially irrelevant to an XML processor. However, a very different situation arises when the two documents are encrypted. Because the sequence of characters in the two documents is different, they will produce a different encrypted result.

I have shown you several examples of how the permitted differences in XML syntax can result in physical XML document files that consist of distinct sequences in contained characters. The widely available encryption routines have no knowledge of XML syntax—why should they since many were created long before XML existed?—and treat the documents, which are equivalent from an XML viewpoint, as two or more distinct documents.

It might seem as if we could simply standardize particular ways of writing certain types of documents and get around the problem, but that won't work. XML processors of various types recognize that certain types of XML syntax are equivalent. For example, we could have a simple XSLT transformation, which would have a source document like the one in Listing 11.11.

This might be transformed so that the name of the author is contained in an attribute of the <Title> element, like that in Listing 11.12.

```
<?xml version='1.0'?>
<!DOCTYPE Book SYSTEM "Book3.dtd">
<Book>
<Title>XPath Essentials</Title>
<Author>Andrew Watt</Author>
<Publisher>John Wiley</Publisher>
</Book>
```

Listing 11.11 A Simple Description of This Book (Book06.xml).

```
<?xml version='1.0'?>
<!DOCTYPE Book SYSTEM "Book4.dtd">
<Book>
<Title Author="Andrew Watt">XPath Essentials</Title>
<Publisher>John Wiley</Publisher>
</Book>
```

Listing 11.12 A Description of This Book, Following a Possible XSLT Transformation (Book07.xml).

Will every XSLT processor use double quotes? Or single quotes on the Author attribute value? We simply don't know in advance. Since, from an XML point of view, it makes no difference whether single quotes or double quotes are used, it is likely that authors of XML processors gave the matter little consideration. Within the setting where the only goal is XML processing using legal syntax, there is no reason why they should give the matter much consideration.

Since XML is increasingly being passed from one XML-aware processor or application to another, it is possible that at some stage during processing, a change could be made in some aspect of the syntax, which could then mean that, when encrypted, the digest of two (or more) possible forms of the document are not the same.

The question therefore arises that if the two digests are not the same, what is the cause for the difference? Is it because the digest has been tampered with en route? Perhaps we might think that we could examine the internals of the encrypted digest to determine what the cause for the difference(s) is. But it would be a poor encryption algorithm that would permit such ad hoc examination of the contents of a digest.

The only practical solution to the problem is that a canonical version exists for a set of XML documents that are logically equivalent. If the canonical version is used as the input to the encryption process, then it will produce a consistent encrypted result. Any XML document that is a member of the set of logically equivalent documents should produce the same canonical version and, therefore, when first converted to that canonical version, it should produce the same encrypted result.

What Does Canonical XML Do?

Canonical XML provides a means to represent XML documents that are logically equivalent in a consistent way. Thus there is one canonical XML document for each set of logically equivalent XML documents offered in the previous section.

The canonical XML representation, since it is always the same for that set of documents, can be the input into an encryption or similar process. With a consistent input, the output encrypted file will also be consistent, to the degree that the encryption algorithm is consistent. In practice the widely accepted encryption algorithms are sufficiently consistent that there is no practical issue.

In most situations, if two XML documents have the same canonical form, then they are equivalent in their application context. There are special case exceptions to that general rule, but they needn't concern us here.

NOTE **It is possible for two XML documents to have different canonical forms but still be equivalent in some specific application contexts. The Canonical XML Recommendation does not attempt to consider that unusual situation.**

What Is Canonicalization?

Canonicalization is the process by which an XML document is converted into its equivalent canonical XML representation.

Broadly, canonicalization can be viewed as a two-step process:

1. An octet stream is converted to an XPath tree (or a node set is selected directly).
2. The XPath node set is converted into an octet stream, which constitutes the canonical form.

NOTE **Canonicalization is sometimes referred to as c14n. If you count the number of letters between the beginning "c" and final "n" of canonicalization, the reason should be apparent.**

Canonicalization, when viewed in detail, is a potentially fairly complex process. The main steps of which are summarized here:

- The XML document is encoded in UTF-8.
- Line breaks are normalized to #xA on input, before XML parsing.
- Attribute values are normalized, as if by a validating processor.
- Character references and parsed entity references are replaced by their replacement text.
- CDATA sections are replaced by their character content.
- The XML declaration and DOCTYPE declaration are removed, since neither is represented in the XPath data model.
- Empty elements are converted to start tag–end tag pairs.
- Whitespace outside the document element and within start and end tags is normalized.
- All whitespace in character content is retained, with the exception of whitespace removed during line feed normalization (described in the second bullet point above).
- The delimiters for attribute values are set to quotation marks (double quotes), not apostrophes.
- Special characters in attribute values or in character content are replaced by character references.

- Superfluous namespace declarations are removed from each element.
- Default attributes, declared in a DTD or XSD schema, are added to their respective elements.
- Lexicographic order is imposed on the namespace declarations and attributes of each element.

XPath, Subsets, and Canonical XML

It has taken us quite some time to get here, but now we need to look at where XPath plays a role in canonical XML.

The data model for Canonical XML is the same data model as that for XPath 1.0, which was described in Chapter 3. The specification indicates that implementations do not need to use an XPath implementation to produce the results, but that the results must be equivalent to using an XPath processor. Thus, I will describe the following as if an XPath processor were used.

Canonicalization can be applied to a whole XML document or it can be applied to a document subset. The means of expressing which subset of the document is to be processed employs XPath expressions.

An XPath expression is used to define a node set. In Canonical XML, nodes present in the node set are included in the canonical form of the XML document subset. A node that is not present in the node set generated by an XPath expression is excluded from the canonical form in question.

The processing algorithm of Canonical XML means that if a node is excluded from a node set then it may still impact on the form of its descendant nodes, by virtue of any namespace declarations that are in scope for it.

The canonicalization process takes two parameters. The first parameter is either an XPath node set or an octet stream containing a well-formed XML document. It is required that the input document be well-formed, but it need not be validated. If the first parameter is an octet stream, then XPath requires that it be converted to XPath by an XML processor, which carries out the following tasks in order:

- Normalize line feeds
- Normalize attribute values
- Replace CDATA sections with their character content
- Resolve character references and parsed entity references

The second parameter of input to the XML canonicalization method is a Boolean value, which indicates whether or not comments in the source XML document are to be included or not in the canonical XML output. If the canonical form of an XML document or subset of such a document contains comments corresponding to the input node set, then the canonical form is called *canonical XML with comments*.

Canonical XML implementations are required to be able to produce canonical form excluding all comments. It is recommended, but not required, that they can also produce canonical XML with comments. When producing canonical XML with comments, it may be important to be aware that the XPath data model does not create comment nodes for comments contained within the DOCTYPE declaration. You may remember that the

XPath data model does not represent either the DOCTYPE declaration or the XML declaration. Thus neither of those parts of an XML document are represented in a canonical form.

NOTE **The XML declaration cannot be represented by the XPath data model; however, the XML declaration is not necessary since the canonical form is invariably encoded in UTF-8. Therefore, the encoding attribute of any XML declaration would serve no purpose. Additionally, the absence of an XML declaration specifically indicates that the version of XML whose canonical form is being represented is XML version 1.0.**

Canonical XML, by virtue of the XPath data model, can represent root, element, namespace, attribute, comment, processing instruction, and text nodes.

Element nodes have attribute nodes that encompass non-namespace declaration attributes present in the start tag of the element, as well as default attributes defined in any accompanying document type definition, DTD, or XSD schema.

XML canonicalization is namespace aware, as you would expect, due to its dependency on the XPath data model. At one time it was envisaged that the process of canonicalization would also involve namespace rewriting (that is, it would achieve a uniform use of namespace prefixes for a given namespace URI). However, the full Recommendation for Canonical XML rejected that methodology since, in certain circumstances, the meaning of a document is changed by namespace rewriting and thus it would no longer reliably be logically equivalent to the input XML document.

An element node has namespace nodes representing all namespaces in scope for the element that the node represents. This creates particular issues for the processing of subdocuments, since it is necessary for the canonicalization processor not only to process the element nodes selected by an XPath expression but also to process all parent nodes, since those may have namespace nodes that also must be associated with the selected element nodes. A true XPath processor would have automatically created namespace nodes for the selected element nodes but, since implementations equivalent to XPath are permitted by the canonicalization specification, it is important to be sure that all namespace nodes have been created correctly.

The Canonical XML Recommendation deprecates the use of relative URIs in namespace URIs. An implementation of canonical XML must report an operation failure if a relative namespace URI is used in a namespace declaration. Thus, not all documents for which an XPath in-memory tree can be created can be processed by a canonicalization implementation.

The XPath data model represents data by the use of the UCS character set. Canonicalization must use XML processors that can use UTF-8 and UTF-16 characters. In addition, support for ISO-8859-1 encoding is recommended. UCS character encodings include UTF-8, UTF-16, UTF-16BE, UTF-16LE, UCS-2, and UCS-4. If a document does not already use UCS encoding, then the XPath processor must convert to UCS using Unicode Normalization Form C (see the specification for further details).

The details of XPath processing differ, depending on whether the first input parameter is an octet stream or a node set. First let's look at the situation where the input parameter is an octet stream reflecting an XML document.

After an XPath node set has been created, by conversion of an octet stream, the XPath processor sets an XPath evaluation context consisting of the following:

- A context node, initialized to the root node of the input XML document
- A context position, initialized to 1
- A context size, initialized to 1
- A library of functions corresponding to the core XPath 1.0 function library
- An empty set of variable bindings
- An empty set of namespace declarations

Once the evaluation context has been established, the default XPath expression applied depends on whether or not comments are to be included.

If, as in normal canonical XML, comments are to be excluded, then the default XPath expression is as follows, which means that all element nodes, all attribute nodes, all namespace nodes, all text nodes, and all comment nodes will be selected but *no* comment nodes will be selected.

```
(//node() | //@* | //namespace::*)[not(self::comment())]
```

If, as in canonical XML with comments, comments *are* to be included, then the default XPath expression is

```
(//node() | //@* | //namespace::*)
```

Then all element nodes, attribute nodes, namespace nodes, comment nodes, text nodes, and processing instruction nodes are to be selected.

If the first input parameter is a node set, then the situation is a little different. If the input is a node set, then the node set must *explicitly* contain all the nodes to be rendered to the canonical form. If we had the following fragment in a source document that we wished to render in canonical form

```
<SomeElement>
<AFirstChild></AFirstChild>
<ASecondChild></ASecondChild>
</SomeElement>
```

it would not be enough to simply use the expression

```
//SomeElement
```

since that would simply select the node representing the <SomeElement> element but would exclude its child nodes. Nor with the following XML fragment would the namespace nodes, which are present on the <XMML:SomeElement> element or those on the child element nodes, be selected.

```
<XMML:SomeElement xmlns:XMML="http://www.xmml.com/SomeSchema/">
<XMML:AFirstChild></XMML:AFirstChild>
<XMML:ASecondChild></XMML:ASecondChild>
</XMML:SomeElement>
```

A similar problem would occur if we used the XPath id ( ) function in an expression id("special") with the following fragment in the relevant XML document.

```
<XMML:SomeElement
  id="special"
  xmlns:XMML="http://www.xmml.com/SomeSchema/">
<XMML:AFirstChild></XMML:AFirstChild>
<XMML:ASecondChild></XMML:ASecondChild>
</XMML:SomeElement>
```

Document Order

The XPath data model defines a node set to be unordered. However, the XPath specification also alludes to the notion of document order, which for element nodes is essentially the sequence in the source document in which the opening angled bracket of the start tag of an element occurs. More generally, document order can be defined as the order in which the first character of the representation of a node in the source XML document occurs, following the expansion of general entities. The definition does not apply to namespace or attribute nodes, whose ordering is application dependent.

In canonical XML the ordering of namespace and attribute nodes is defined according to the following rules:

- An element's attribute and namespace nodes take a later (greater) position in document order than the element itself but an earlier (lesser) position than the element's child nodes.
- Namespace nodes have an earlier (lesser) position in document order than attribute nodes.
- An element's namespace nodes are sorted lexicographically (alphabetically) by the local name. The default namespace node, if there is one, has no local name and is treated as lexicographically least (that is, earliest in document order).
- An element's attribute nodes are sorted lexicographically with the namespace URI as the primary key for the sort and the local name as the secondary key. An empty namespace URI is treated as lexicographically least (that is, earliest in document order).

Lexicographic ordering, technically speaking, orders strings from the least to greatest alphabetically, based on UCS codepoint values, which is equivalent to lexicographic ordering in UTF-8. In practice, if the relevant names are in English, then normal alphabetical ordering will informally describe the resultant order.

Canonical XML makes use of the concept of document order when processing the node-set selected by an XPath expression for elements as described in XPath 1.0 and for attribute and namespace nodes as just described.

The Final Step

The final step in the production of the canonical form is conversion of the node set into an octet stream. Each node in the node set results in the generation of the representative

UCS characters for that node, each node being processed in document order. The stream of UCS characters is then encoded in UTF-8.

Each node in the node set is processed only once. Processing of an element node is complete only when processing of all other nodes for which it is an ancestor has been completed. In other words, the processing of an element node also involves the processing of any child nodes, namespace nodes, or attribute nodes that are contained in the node set. Remember that any child nodes not selected in the XPath expression that results in the node set are not processed when the canonical form is generated.

Once an element is fully processed (including processing of any child, namespace, or attribute nodes), that element node is removed from the node set.

If an element node is processed and its ancestor nodes are not included in the node set, the ancestor nodes are nonetheless examined in order to determine whether or not they have any associated namespace URIs, which also are in scope for their descendants.

If a node is in the node set, text may be generated to correspond to that node. The rules for generation of text depend on which of the seven allowed types of XPath node is being processed. The following lists the rules for generation of text from each of the seven XPath node types:

Root node. No text is generated. There is no XML declaration or any DOCTYPE declaration generated.

Element node. If an element node is in the node set, then the generated result is as follows: an opening angled bracket, "<", followed by the QName of the relevant element, followed by the result of processing the namespace axis, followed by the result of processing the attribute axis, a closing angled bracket, ">", followed by the result of processing in document order the child nodes of the element node that are present in the node set, followed by an opening angled bracket, a "/" character, the QName of the element which the element node represents, and, finally, a closing angled bracket.

Namespace nodes. If the parent node of the namespace node's parent element node has a namespace node with the same local name and value, then the first namespace node is ignored. That makes sense since the namespace declaration should be on the outermost element whose element node is the parent of the respective namespace node. If the parent node of a namespace node's parent element node does *not* have a namespace node with the same local name and value, then an appropriate namespace declaration is added within the start tag of the element represented by the parent node of the namespace node.

Attribute nodes. This generates a space, the attribute node's QName, an equals sign, a double quote (remember the canonical form uses only double quotes), the attribute value, and a closing double quote. The string value of the node is modified by replacing all ampersands by &, all open angled brackets with <, all quotation mark characters with ", and the whitespace characters #x9, #xA, and #xD with character references. The character references are written in uppercase hexadecimal with no leading zeroes.

Text nodes. These generate the string value of the text node, except that it is modified by replacing all ampersands with &, all opening angled brackets with

<, all closing angled brackets by >, and all #xD characters are replaced by
.

Processing instruction nodes. A processing instruction node produces the opening processing instruction symbol, "<?", followed by the PI target name of the node, followed by a space (if the string value is not empty), then followed by the string value of the processing instruction. If it has one, it is followed by the closing processing instruction symbol, "?>".

Comment nodes. If canonical XML (without comments) is being generated, then comment nodes generate no text. If canonical XML *with* comments is being generated, then the opening comment symbol, "<!—", is followed by the string value of the comment node, followed by the closing comment symbol, "—>".

As you can see, the text output from each node is very precisely defined. It has to be. If there are any differences between characters in the canonical form due to inconsistencies or ambiguities in the generated text, the canonical form may differ, but will be only spuriously different. It is therefore of great importance that each node in the node set is processed according to its node type and its position in document order if the canonical form is to be wholly predictable, as it must be to serve its purpose.

Document Subsets

I indicated earlier that the parent and ancestor nodes of nodes within the node set are examined during processing. The ancestor axis of a node in the node set is examined to find the nearest ancestor node that possesses attribute nodes representing xml:lang and xml:space attributes, since the value of these attributes on an ancestor node is applicable to the node that is in the node set too. Any such nodes that are found are discarded if the same attribute node is found in the attribute axis of the node in the node set. What that means is that if an ancestor has an xml:space attribute, for example, but the node in the node set also possesses an xml:space attribute node, it is only the attribute node on the element that is actually in the node set which applies. If an attribute node is present on an ancestor node but not present in the attribute axis of the node in the node set, then the attribute node of the ancestor element node is added to the nodes in the attribute axis of the element in the node set.

Well-Formed

The canonical form of a complete XML document is always well-formed. The canonical form of an XML document subset may not be well-formed. For example, there may be two sibling elements rather than a single element that might be called the "document element" of the document subset.

While it is not a requirement of the Canonical XML specification that the canonical form of XML document subsets be well-formed, it is likely in practice that they will typically be well-formed. One important practical reason is that the canonical form of an XML document or document subset will frequently be intended for further XML processing. If the canonical form is not well-formed, then an error can be expected during any attempt at XML processing of the canonical form.

If a canonical form is well-formed, then further application of the canonicalization algorithm to that canonical form will produce no further change in its physical representation.

Some Canonicalization Examples

In the previous sections, I had to compress abstract ideas to fit the concepts and steps of processing into a reasonable space. In this section, I will show you some simple examples to illustrate the "before" and "after" of the canonicalization process.

Let's take a simple Hello, World! document like that in Listing 11.13.

If we process to produce canonical XML (without comments), the canonical form will look like Listing 11.14.

Notice that all the comments are absent from the canonical form. The XML declaration and the DOCTYPE declaration are absent. The delimiters for the value of the type attribute on the <document> element have been changed from apostrophes to quotation marks. Extraneous whitespace within the <?xml-stylesheet?> and <?pi-without-data?> processing instructions has been removed in accordance with the procedures described earlier.

The canonical form with comments (see Listing 11.15) demonstrates the changes just described with the exception that comments present in the input document remain in the canonical form with comments.

The following example illustrates the use of canonicalization of XML document subsets and is taken from the Canonical XML Recommendation.

Let's say we have an input XML document like that in Listing 11.16.

TEAMFLY

```
<?xml version="1.0"?>
<?xml-stylesheet   href="document.xsl"   type="text/xsl"   ?>
<!DOCTYPE doc SYSTEM "document.dtd">
<document type='simple'>Hello, world!<!-- This is a trivial comment. --
></document>
<?pi-without-data     ?>
<!-- This is a comment outside the document element -->
```

Listing 11.13 A Greeting Expressed in XML (HelloWorld.xml).

```
<?xml-stylesheet href="document.xsl" type="text/xsl"?>
<document type="simple">Hello, world!</document>
<?pi-without-data?>
```

Listing 11.14 The Canonical Form of Listing 11.13 without Comments (HelloWorldWithoutComments.xml).

```
<?xml-stylesheet href="document.xsl" type="text/xsl"?>
<document type='simple'>Hello, world!<!-- This is a trivial comment. -->
</document>
<?pi-without-data?>
<!-- This is a comment outside the document element -->
```

Listing 11.15 The Canonical Form of Listing 11.13 with Comments (HelloWorldWithComments.xml).

```
<?xml version='1.0'?>
<!DOCTYPE doc [
<!ATTLIST e2 xml:space (default|preserve) 'preserve'>
<!ATTLIST e3 id ID #IMPLIED>
]>
<doc xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org">
   <e1>
      <e2 xmlns="">
         <e3 id="E3"/>
      </e2>
   </e1>
</doc>
```

Listing 11.16 An XML Document with Nested Elements (Doc.xml).

We want to apply the following XPath expression to it:

```
<!— Evaluate with declaration xmlns:ietf="http://www.ietf.org" —>

(//node() | //@* | //namespace::*)
[
self::ietf:e1 or (parent::ietf:e1 and not(self::text() or self::e2))
or
count(id("E3")|ancestor-or-self::node()) = count(ancestor-or-
self::node())
]
```

Let's examine what the XPath expression means. The first part of the expression indicates that any node type will be included in the document subset, but those nodes are further filtered by the predicate. The predicate indicates that nodes may be included if they satisfy any of the following criteria:

- A node that represents an <ietf:e1> element.
- A node that has an <ietf:e1> element node as a parent, but which itself is neither a text node nor an element node representing an <ietf:e2> node.
- The results returned by the id ( ) function and the count ( ) function nested within the outer count ( ) function cause a value of true to be returned.

The resulting canonical form of the document subset is as follows:

```
<e1 xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org"><e3
  xmlns="" id="E3" xml:space="preserve"></e3></e1>
```

Exclusive XML Canonicalization

Canonical XML provides a means to serialize an XML subdocument so that it retains its namespace and some other XML context. In certain situations it may be necessary to achieve the opposite and to remove, as far as possible, the context information. For example, it might be necessary to place an XML signature (see the following section) over an XML subdocument so that the signature doesn't break when the subdocument is removed from its original context or inserted into a different document. To achieve that as much contextual information as possible must be removed.

NOTE At the time of writing, the Exclusive XML Canonicalization specification is at first Working Draft stage, so it is likely that details will change. Refer to www.w3.org/TR/xml-exc-c14n for the current version of the specification.

Why is the Exclusive XML Canonicalization specification relevant to XPath? The Exclusive XML Canonicalization specification defines a particular way of serializing an XPath node set.

The Exclusive XML Canonicalization specification is dependent for some of its concepts and terminology on the Canonical XML Recommendation described earlier and, of course, is also dependent on the XPath 1.0 Recommendation.

Exclusive XML Canonicalization is designed for use with XML Signatures, to be described in the next section. Exclusive XML Canonicalization is one of the canonicalization methods that can be specified within a <signedInfo> element of XML Signatures.

A common situation where Exclusive XML Canonicalization could be used is in association with the use of XML protocols. A message being transmitted in the context of an XML protocol is typically wrapped in layers of information, which may indicate the destination of the message, etc. These wrapping layers of XML alter the context of the canonical form. Since the content of a message may be passed on to another XML processor or application, it is important that any digital signature information is not rendered void by the change in context. Thus, a way to digitally sign a canonical form while removing, as far as possible, any dependency on its XML context is important for the use of XML protocols.

Both Canonical XML and Exclusive Canonical XML are intended to be used with XML signatures, so let's take a moment to see what the use of XPath in canonicalization algorithms is intended to support.

XML Signatures

The presence of a signature on a paper document, and the authenticity of that signature, can have a significant impact on our lives. When a signature is present on paper we can

typically authenticate it by a variety of personal or visual cues. In the context of electronic commerce and data interchange, confirmatory or personal cues are typically absent. The burden of authentication has to be electronic. Given the increasing use of the Internet for business transactions, sometimes of substantial value, it is important that a secure and reliable mechanism exist for establishing the authenticity of documents. The XML Signatures Syntax and Processing specification aims to provide a reliable means of representing a digital signature on arbitrary content in XML syntax.

XML Signatures will provide the means to authenticate a document, a reassurance of data integrity for the content of the signed data, and a basis for nonrepudiatability.

A digital signature, including XML signatures, makes use of cryptography to transform some data into an unintelligible form and, at the appropriate time, to make the reverse transformation to make the data intelligible. To provide sufficient security for practical use, the mathematical algorithms used in encryption are of significant complexity. Typically they use a secret piece of information, called a *key*, known to the sender to provide encryption that resists unauthorized decryption. Often two keys will be used, a *public* key, which is made available to anyone, and a *private* key, which only the sender should know. This is called *asymmetric key encryption*, since the private key is used by one party and a different key—the public key—is used by the other. If the person owning the private key encrypts a message, then anyone using the public key to decrypt it can be confident that it was that party who created the message. Similarly, if someone wishes to send a confidential message to the owner of the private key, the message can be encrypted using the public key in the expectation that only the owner of the private key will be able to decrypt it.

XML signatures can use a variety of encryption technologies. It also uses canonicalization described earlier in this chapter.

An XML signature would look something like this:

```
<Signature Id="ABasicSignature"
  xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
   <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-
     c14n-20010315"/>
   <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#
     dsa-sha1"/>
   <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
     <Transforms>
       <Transform Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-
         20010315"/>
     </Transforms>
     <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
    <DigestValue>JiDa35970dD1az03kLmdEw...</DigestValue>
   </Reference>
 </SignedInfo>
 <SignatureValue>DB32xi88dDGfa8315...</SignatureValue>
  <KeyInfo>
    <KeyValue>
      <DSAKeyValue>
        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
```

```
        </DSAKeyValue>
      </KeyValue>
    </KeyInfo>
  </Signature>
```

The namespace declaration in the start tag of the <Signature> element is the namespace used by the April 2001 Candidate Recommendation. The namespace in the final Recommendation may differ from that.

If you look at the content of the <Transforms> element, you will see that the <Transform> element has an Algorithm attribute with a value of "http://www.w3.org/TR/2001/REC-xml-c14n-20010315", meaning that the algorithm used is the canonicalization algorithm we looked at earlier in the chapter.

Using Canonical XML with XML Signatures

Let's take a brief look at how Canonical XML works with digital signatures.

The sending party would create an XML signature using a stated algorithm for a particular XML document or document subset. The signature and the signed document would be sent to the receiving party. The receiving party could then calculate a signature digest on the received version of the document. If the signature digest for the sent version of the document matches the signature digest for the received version of the document, then there is a very high degree of confidence that no logical changes have been made to the canonical forms of the sent document (or document subset) or the received document (or document subset). Since it is the canonical XML versions that are being compared, syntax changes that do not affect the logical meaning (such as the syntax variants demonstrated earlier in this chapter) are not relevant. Thus the receiver of a message can be confident that it has not been tampered with during transit.

As indicated earlier, the <Transform> element and its algorithm attribute are used to associate the Canonical XML algorithm with a particular XML signature.

XPath Transforms in XML Signatures

The <Transforms> element may contain several <Transform> elements. A <Transform> element may, as well as indicating the use of the canonicalization algorithm, make use of XPath and XSLT transforms to allow the signer to derive an XML document that uses only selected portions of the input XML document.

XPath Filtering in XML Signatures

An XPath filter may be defined in an <XPath> element, which is a child element of the <Transform> element.

```
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
   <CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-
     c14n-20010315"/>
```

```
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#
      dsa-sha1"/>
     <Reference URI="">
       <Transforms>
         <Transform
           Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
           <XPath xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
           not(ancestor-or-self::dsig:Signature)
           </XPath>
         </Transform>
       </Transforms>
       <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
       <DigestValue></DigestValue>
     </Reference>
   </SignedInfo>
   <SignatureValue></SignatureValue>
</Signature>
```

The XPath filter shown above omits all <dsig:Signature> elements and their descendants.

The here ( ) function allows the selective omission of the <dsig:Signature> element, which contains the <XPath> element, as shown in the following code:

```
<XPath xmlns:dsig="&dsig;">
count(ancestor-or-self::dsig:Signature |
here()/ancestor::dsig:Signature[1]) >
count(ancestor-or-self::dsig:Signature)</XPath>
```

In this section, I have given you a very brief introduction as to how XPath will be used in XML Signatures. For more information, consult the XML Signatures specification located at www.w3.org/TR/xmldsig-core/.

Language-Specific Implementations

The Java Community Process is often an early indicator of developments in the arena of the Java language by Sun Microsystems. At the time of writing, two distinct, but inevitably interrelated JSRs (Java Specification Requests) have proposed implementation of the XML Signatures and the XML Encryption specifications. The XML Digital Signatures JSR is located at http://jcp.org/jsr/detail/105.jsp and the XML Encryption JSR at http://jcp.org/jsr/detail/106.jsp.

XACML and XPath

The Extensible Access Control Markup Language is at a very early stage of development at the time of writing. However, the charter for the OASIS Technical Committee, which is developing XACML, indicates that XPath is likely to be the means of binding XACML to existing protocols, although that has yet to be confirmed.

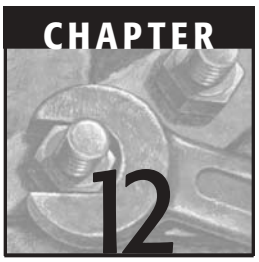
The few preliminary documents accessible at the time of writing are likely to be rapidly updated. A number of relevant use cases are being examined, for example, relating to access to medical information or to information exchanged in the context of ebXML (see www.ebxml.org). Because of the draft nature of those documents they are not presented here. Current documents will be available using links from the main XACML Web page, www.oasis-open.org/committees/xacml/, or if you wish to follow developments in detail, the archive of the XACML mailing list provides a useful and up-to-the-minute view of progress. The mailing list archives are located at http://lists.oasis-open.org/archives/xacml/.

The Access Control Technical Committee hopes to have a final version of the XACML specification ready to be considered by the OASIS membership by March 2002.

Looking Ahead

This chapter touched on uses of XML, which will be critical as the XML-based Web becomes more and more pervasive. The precise use of XPath in XML Signatures may change, but the fact that XPath is a basis for both XML Canonicalization and of XML Signatures means that XPath is becoming more widely used.

Chapter 12 shows you further practical examples for selecting elements and attributes using XPath.



Selecting Elements and Attributes

Selecting elements and attributes is one of the most common uses of XPath. This chapter will provide you with practical examples and guidance for selecting elements and attributes using XPath. You can match these examples directly to tasks that you have in hand to help you master and complete them quickly and efficiently.

If you have been thrown into an XSLT/XPath project without much understanding of XPath, have very limited understanding of XPath, and you have skipped most of this book, then this chapter is for you. It provides examples that demonstrate basic techniques to get you up and running and producing working code, even though you may not yet fully understand how XPath works or all the jargon. You should be able to find sections labeled in clear English that describe programming tasks you need to get done and show you how to use XPath to make commonly used selections of elements and attributes.

If you have been reading the book chapter by chapter from the beginning, you will probably find little new here, other than further examples of how to use XPath to select elements and attributes from XML source documents to process in whatever way you choose.

The examples in this chapter will all use XPath with XSLT.

Selecting Elements

The first part of the chapter will address the selection of elements using XPath.

Selecting Elements by Name

Selecting elements by name is one of the most basic operations using XPath. We simply express the name of the element node that represents the element at an appropriate place within an XPath location path.

This is demonstrated in Listing 12.1.

To select an element, we simply use the element name within an XPath location path. The following directly selects all <Drink> elements in the document, which happen also to be children of <Drinks> elements, using the abbreviated absolute syntax:

```
/SimpleDoc/Drinks/Drink
```

One of the most straightforward uses of such an absolute location path is in the <xsl:apply-templates> element in the main template of an XSLT stylesheet, such as that in Listing 12.2.

The <xsl:apply-templates> element in the main template specifies by means of its select attribute that <Drink> elements, which are children of <Drinks> elements, which, in turn, are children of a <SimpleDocs> element, which, in turn, is a child of the root node, will be processed. The match attribute of the <xsl:template> element defines what

```
<?xml version='1.0'?>
<SimpleDoc type="WonderRestaurants Menu">
<Foods>
<Food>Hamburger</Food>
<Food>French Fries</Food>
<Food>Pizza</Food>
<Food>Aberdeen Angus Steak</Food>
</Foods>
<Drinks>
<Drink>Cola</Drink>
<Drink>Root beer</Drink>
<Drink>Seven Up</Drink>
<Drink>Coffee</Drink>
</Drinks>
</SimpleDoc>
```

Listing 12.1 A Simple Menu Expressed in XML (SimpleDoc.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
```

Listing 12.2 An XSLT Stylesheet to Select Drinks Available in the Menu Shown in Listing 12.1 (SimpleDoc.xsl).

```
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements by name</title>
</head>
<body>
<h3>We will select the &lt;Drink&gt; elements by name.</h3>
<xsl:apply-templates select="/SimpleDoc/Drinks/Drink"/>
</body>
</html>
</xsl:template>

<xsl:template match="Drink">
<p>The &lt;Drink&gt; element in position <xsl:value-of
  select="position()"/> contains <xsl:value-of select="."/>.</p>
</xsl:template>

</xsl:stylesheet>
```

Listing 12.2 (*Continued*)

the template will instantiate for each such <Drink> element selected in the main template. The output is shown in Figure 12.1.

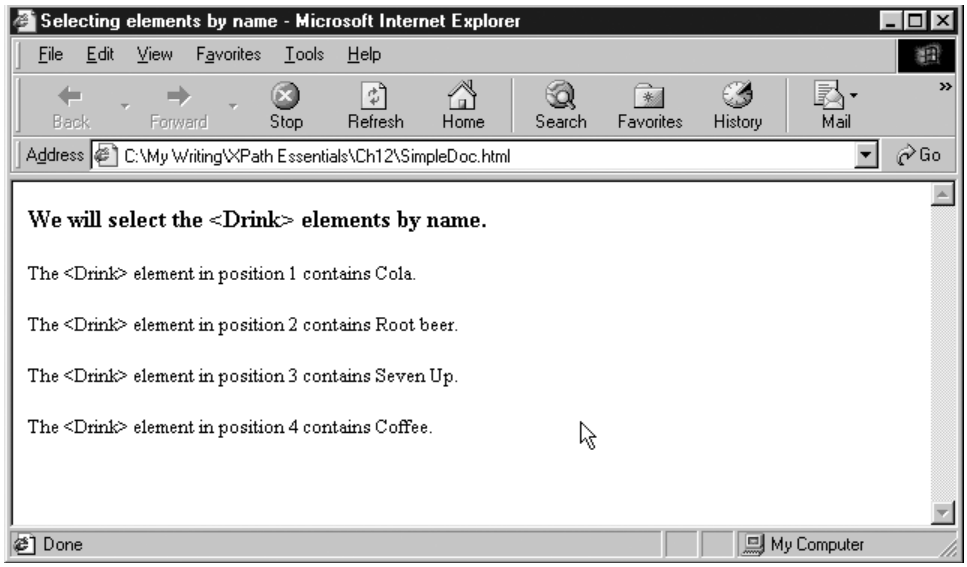

Selecting elements by name - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\SimpleDoc.html
Go
We will select the <Drink> elements by name.
The <Drink> element in position 1 contains Cola.
The <Drink> element in position 2 contains Root beer.
The <Drink> element in position 3 contains Seven Up.
The <Drink> element in position 4 contains Coffee.
Done
My Computer

Figure 12.1 Selecting <Drink> Elements by Name.

Selecting Elements by Parent Characteristics

A particularly common characteristic of a <Parent> element, which is used in selecting an element, is the name of the <Parent> element. We saw this in Listing 12.2, where the select attribute of the <xsl:apply-templates> element looked like this:

```
<xsl:apply-templates select="/SimpleDoc/Drinks/Drink"/>
```

The <Drink> element would be processed only if the element type name of its <Parent> element was "Drinks".

Selecting Elements by Value

You may want to select elements that have a particular value contained within them. Listing 12.3 is a source document we can use to explore that.

The stylesheet in Listing 12.4 shows how to select elements by virtue of their content.

TEAMFLY

```
<?xml version='1.0'?>
<Travel>
<Flight>
<Person>Peter Gray</Person>
<From>New York</From>
<To>London</To>
<FlightNo>ABC123</FlightNo>
<Class>Business</Class>
<Date>2002-08-11</Date>
</Flight>
<Flight>
<Person>Karen Smith</Person>
<From>Tokyo</From>
<To>Paris</To>
<FlightNo>TKY199</FlightNo>
<Class>Business</Class>
<Date>2003-04-09</Date>
</Flight>
<Flight>
<Person>Carol Peters</Person>
<From>San Francisco</From>
<To>New York</To>
<FlightNo>DBE890</FlightNo>
<Class>Economy</Class>
<Date>2002-02-04</Date>
</Flight>
<Flight>
```

Listing 12.3 A Catalog of Travel Expressed in XML (Travel.xml).

```
<Person>Peter Gray</Person>
<From>London</From>
<To>Paris</To>
<FlightNo>BAC876</FlightNo>
<Class>Economy</Class>
<Date>2001-12-23</Date>
</Flight>
<Flight>
<Person>Chloe Jamna</Person>
<From>Rome</From>
<To>Toronto</To>
<FlightNo>ACD789</FlightNo>
<Class>Business</Class>
<Date>2002-08-31</Date>
</Flight>
</Travel>
```

Listing 12.3 (*Continued*)

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements by content</title>
</head>
<body>
<h3>An example of selecting elements by content - </h3>
<xsl:apply-templates select="Travel/Flight/Person[.='Peter Gray']"/>
</body>
</html>
</xsl:template>

<xsl:template match="Person">
<h3>Flight for <xsl:value-of select="."/>:</h3>
<p><b>Trip from: </b> <xsl:value-of select="../From"/></p>
<p> <b>Trip to: </b> <xsl:value-of select="../To"/></p>
```

Listing 12.4 A Stylesheet to Select Elements on the Basis of Element Content (Travel.xsl). (*continues*)

```
<p><b>Class: </b> <xsl:value-of select="../Class"/></p>
<p><b>Date: </b> <xsl:value-of select="../Date"/></p>
<br />
</xsl:template>
</xsl:stylesheet>
```

Listing 12.4 (*Continued*)

It is the <xsl:apply-templates> element in the main template that selects for processing elements which have as their content the string, or text node, "Peter Gray".

```
<xsl:apply-templates select="Travel/Flight/Person[.='Peter Gray']"/>
```

The predicate `[.='Peter Gray']` means that only Person element nodes that have the content "Peter Gray" will have a template instantiated. Thus when the XSLT processor matches on the following code, it is only those Person element nodes with value of "Peter Gray" that are then processed.

```
<xsl:template match="Person">
```

Notice in the Person template that most of the <xsl:value-of> elements have a select attribute of this form, `"../Date"`, which means that it is the Date element child node(s) of the parent node of the context node (at that time a Person element node) that is processed.

Selecting and Sorting Elements by Value

Commonly you will not only want to select elements but also sort them in some way before presenting them or using them.

The following stylesheet (Listing 12.5) will select the <Food> elements from Listing 12.1 and present them on screen sorted in alphabetical order according to the value of the content of the <Food> element.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
```

Listing 12.5 A Stylesheet to Sort <Food> Elements (SortedFood.xsl).

```
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting and sorting elements by value</title>
</head>
<body>
<h3>This will display the &lt;Food&gt; elements sorted in alphabetical
  order.</h3>
<xsl:apply-templates select="/SimpleDoc/Foods/Food">
<xsl:sort select="."/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Food">
<p><xsl:value-of select="."/></p>
</xsl:template>

</xsl:stylesheet>
```

Listing 12.5 (*Continued*)

Notice in the main template that the <xsl:apply-templates> element has an <xsl:sort> element nested within it:

```
<xsl:apply-templates select="/SimpleDoc/Foods/Food">
 <xsl:sort select="."/>
</xsl:apply-templates>
```

The select attribute of the <xsl:apply-templates> element works as normal selecting Food element nodes, which are child nodes of a Foods element node, which, in turn, is a child node of a SimpleDoc element node, which, in turn, is a child of the root node. The nested <xsl:sort> element indicates that a sort is carried out. The select attribute of the <xsl:sort> element indicates that it is sorted on ".", which of course is the abbreviated form of `self::node()`. This means that it is the content or value of the context node that is used to carry out the sort.

Thus when the XSLT processor comes to instantiate the <xsl:template> element that matches on a Food element node, then those nodes are processed according to the order of their content. As you can see in Figure 12.2, the values of the content of the <Food> elements are displayed in alphabetical order.

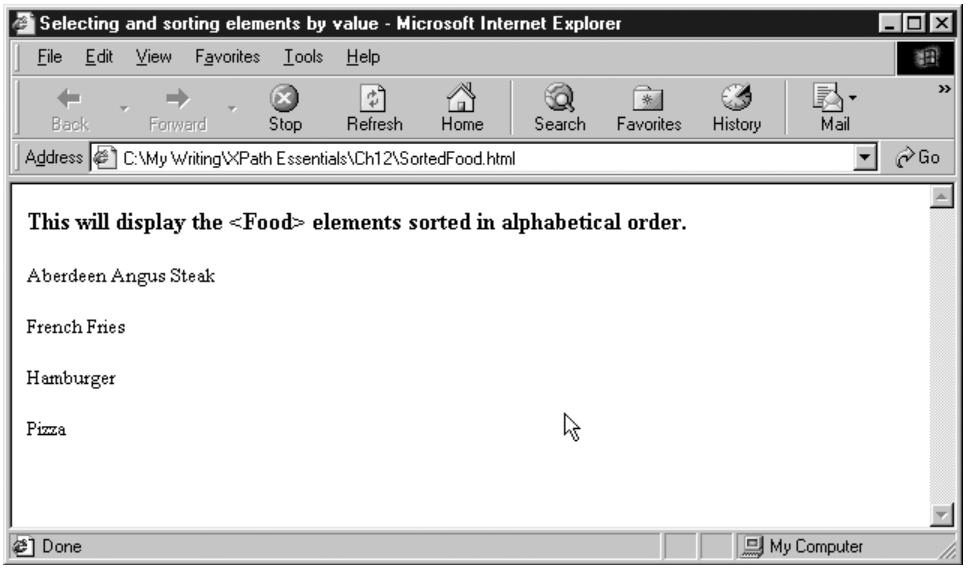

Selecting and sorting elements by value - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\SortedFood.html
Go
This will display the <Food> elements sorted in alphabetical order.
Aberdeen Angus Steak
French Fries
Hamburger
Pizza
Done
My Computer

Figure 12.2 Displaying the Alphabetically Sorted Foods.

Selecting Elements by Position

A common task is to select elements by their position in a group of similarly named elements.

If our XML source document is similar to the one shown in Listing 12.6, we can select elements that are in particular positions in document order.

Let's suppose we simply want to select elements by position and we want to choose the first three countries. For the moment we will simply assume that the <Country> elements are in the desired order. The stylesheet in Listing 12.7 will process our source document and output the first three <Country> elements.

```
<?xml version='1.0'?>
<Countries>
<Country Capital="Washington">USA</Country>
<Country Capital="London">United Kingdom</Country>
<Country Capital="Paris">France</Country>
<Country Capital="Berlin">Germany</Country>
<Country Capital="Ottawa">Canada</Country>
<Country Capital="Canberra">Australia</Country>
<Country Capital="Tokyo">Japan</Country>
</Countries>
```

Listing 12.6 A List of Countries and Their Capital Cities (Countries.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting Elements by Position</title>
</head>
<body>
<h3>Selecting the first three elements.</h3>
<xsl:apply-templates select="/Countries/Country[position() &lt; 4]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Country">
<p>The country in position <xsl:value-of select="position()"/> is
  <b><xsl:value-of select="."/></b>
and its capital is <b><xsl:value-of select="@Capital"/></b>.</p>
</xsl:template>

</xsl:stylesheet>
```

Listing 12.7 A Stylesheet to Select Elements by Name and Position (Countries.xsl).

The way in which we selected only the first three <Country> elements was the <xsl:apply-templates> element in the main template where the value of the select attribute was `/Countries/Country[position() < 4]`. We selected the <Country> elements, which were children of the <Countries> element. The predicate `[position() < 4]` selected those Country element nodes whose position was less than 4. Notice that we had to escape the less than sign by using `<`. XML does not permit unescaped "<" signs within attribute values.

Suppose we wanted to select only those countries in positions 3 to 5. We can achieve that using Listing 12.8. We need to look at the code fairly carefully since there is a subtle trap, which we must be careful to avoid.

We use the <xsl:apply-templates> element in the main template in order to select for matching only those Country element nodes in a position greater than 2. So at that stage we have selected for processing Country element nodes from position 3 and upward.

```
<xsl:apply-templates select="/Countries/Country[position() > 2]"/>
```

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting Elements by Position</title>
</head>
<body>
<h3>Selecting the first three elements.</h3>
<xsl:apply-templates select="/Countries/Country[position() > 2]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Country">
<xsl:if test="position() &lt; 4">
<p>The country in position <xsl:value-of select="position()"/> is
  <b><xsl:value-of select="."/></b>
and its capital is <b><xsl:value-of select="@Capital"/></b>.</p>
</xsl:if>
</xsl:template>

</xsl:stylesheet>
```

Listing 12.8 A Stylesheet to Select Countries Using Two Criteria Relating to Position (Countries02.xsl).

In the <xsl:template>, which matches on Country element nodes, we insert an <xsl:if> element, shown here.

```
<xsl:if test="position() &lt; 4">
```

You may, at first sight, be puzzled by the fact that the value of the test attribute of the <xsl:if> element is that `position() < 4`, but we need to pause and consider what the position is being measured within. Only those Country element nodes after position 2 were selected for processing by the <xsl:apply-templates> element. It is within that ordered node set that the position will now be counted. The nodes in that node set were originally in positions 3, 4, 5, 6, and 7 but in the new node set selected by the <xsl:apply-templates> element they are in positions 1, 2, 3, 4, and 5. Thus if we want to

select the Country element nodes up to and including what was originally position 5 inclusive, we must choose only the first three of the nodes selected for instantiation, and so we have `position() < 4` as the value of the test attribute.

We might want to select the last in a set of documents, as in Listing 12.9.

We must be careful to distinguish whether we mean the last element node in document order or whether we mean the last element node according to some content it contains.

To select the last in a series, if we are sure that they are in date order, we can simply use the XPath last ( ) function, as in Listing 12.10.

```
<?xml version='1.0'?>
<Maintenance>
<MaintVisit>
<Date>2001-07-31</Date>
<Engineer>Jim Wyatt</Engineer>
<Check level="3"></Check>
<PartReplaced>None</PartReplaced>
<Comments>All looks good.</Comments>
</MaintVisit>
<MaintVisit>
<Date>2001-12-25</Date>
<Engineer>Jeffrey Archer</Engineer>
<Check level="1"></Check>
<PartReplaced>SK123</PartReplaced>
<Comments>More wear than expected. Schedule next maintenance for 3
  months.</Comments>
</MaintVisit>
<MaintVisit>
<Date>2002-03-25</Date>
<Engineer>Martin Capel</Engineer>
<Check level="1"></Check>
<PartReplaced>None</PartReplaced>
<Comments>Satisfactory at present.</Comments>
</MaintVisit>
<MaintVisit>
<Date>2002-09-26</Date>
<Engineer>Coral Spassky</Engineer>
<Check level="2"></Check>
<PartReplaced>DB88</PartReplaced>
<Comments>Wear noticeable in DB88. Part replaced.</Comments>
</MaintVisit>
</Maintenance>
```

Listing 12.9 A Simple Maintenance Database Expressed in XML (Maintenance.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting Elements by last() Position</title>
</head>
<body>
<h3>Selecting the last maintenance report.</h3>
<xsl:apply-templates select="/Maintenance/MaintVisit[last()]"/>
</body>
</html>
</xsl:template>

<xsl:template match="MaintVisit">
<p>The most recent maintenance visit took place on <b><xsl:value-of
  select="Date"/></b> and was conducted by
<b><xsl:value-of select="Engineer"/></b>.</p>
<p>The check of the plant was conducted at <b>Level <xsl:value-of
  select="Check/@level"/></b>.</p>
<p>Parts replaced: <b><xsl:value-of select="PartReplaced"/></b>.</p>
<p>Engineer's comments: <xsl:value-of select="Comments"/></p>

</xsl:template>

</xsl:stylesheet>
```

Listing 12.10 A Stylesheet to Select the Last of a Particular Named Element Node (Maintenance.xsl).

The selection for processing is carried out by the <xsl:apply-templates> element in the main template.

```
<xsl:apply-templates select="/Maintenance/MaintVisit[last()]"/>
```

The presence of the predicate, `[last()]`, makes sure that it is only the last of the MaintVisit element nodes in document order that is processed.

Selecting a Preceding Element

In a number of situations you may want to select a preceding element within the same document. There are two XPath techniques that are relevant to this. To demonstrate these, we will use Listing 12.11 as our source document.

First let's look at how we can use the XPath preceding axis to select the immediately preceding element (see Listing 12.12).

```
<?xml version='1.0'?>
<Manual>
<Introduction>
<Paragraph>Para 1 in the Introduction</Paragraph>
<Paragraph>Para 2 in the Introduction</Paragraph>
<Paragraph>Para 3 in the Introduction</Paragraph>
<Paragraph>Para 4 in the Introduction</Paragraph>
</Introduction>
<MainText>
<Paragraph>Para 1 in the main text</Paragraph>
<Paragraph>Para 2 in the main text</Paragraph>
<Paragraph>Para 3 in the main text</Paragraph>
<Paragraph>Para 4 in the main text</Paragraph>
<Paragraph>Para 5 in the main text</Paragraph>
<Paragraph>Para 6 in the main text</Paragraph>
<Paragraph>Para 7 in the main text</Paragraph>
</MainText>
</Manual>
```

Listing 12.11 A Simplified Manual Expressed in XML (Manual.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
```

Listing 12.12 A Stylesheet to Select a Preceding Element (Manual.xsl).

(*continues*)

```
<title>Selecting a preceding element</title>
</head>
<body>
<h3>This example selects a preceding &lt;Paragraph&gt; element. </h3>
<xsl:apply-templates select="/Manual/MainText/Paragraph[3]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<p>This paragraph contains: "<xsl:value-of select='.'/>".</p>
<p>The paragraph before this one contains: "<xsl:value-of
  select='preceding::Paragraph[position()=1]'/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.12 (*Continued*)

The <xsl:apply-templates> element in the main template causes the <xsl:template> matching on a Paragraph element node to be instantiated with the third Paragraph element node, which is a child of a MainText element node as the context node.

```
<xsl:apply-templates select="/Manual/MainText/Paragraph[3]"/>
```

Within that template we first output the content of the context node using the following code.

```
<p>This paragraph contains: "<xsl:value-of select='.'/>".</p>
```

We then use the XPath preceding axis to output the content of the immediately preceding Paragraph node.

```
<p>The paragraph before this one contains: "<xsl:value-of
  select='preceding::Paragraph[position()=1]'/>".</p>
```

The output of the stylesheet is shown in Figure 12.3.

If we had omitted the predicate [position ( )=1] from the select attribute of the <xsl:value> element above (see Listing 12.13), the output is as shown in Figure 12.4.

Another possibility is that you might want to select a preceding sibling of the context node. Listing 12.14 shows how to select the second preceding sibling, with a context node of the fourth Paragraph element node in the Introduction.

The <xsl:apply-templates> element defines the context node for a template to be instantiated as the fourth of the Paragraph element nodes in the introduction. The <xsl:value> element in the following code selects for display the second <Paragraph> element in the preceding-sibling axis.

```
<xsl:value-of select='preceding-sibling::Paragraph[2]'/>
```

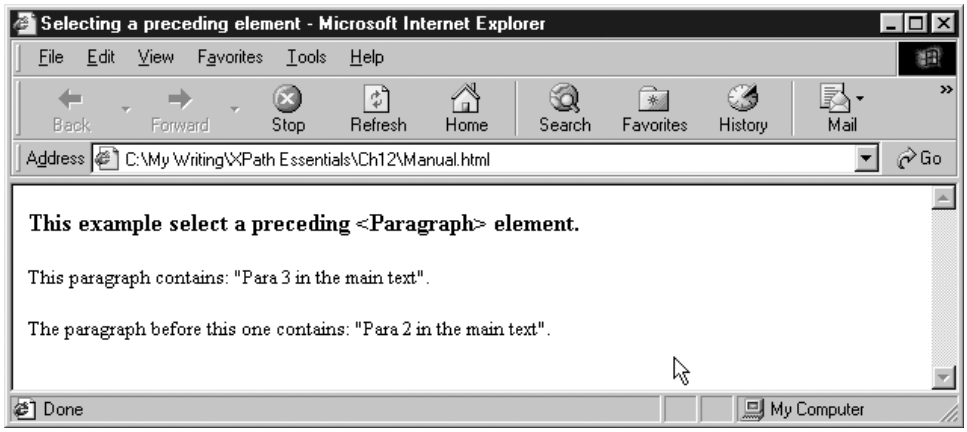

Selecting a preceding element - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\Manual.html
Go
This example select a preceding <Paragraph> element.
This paragraph contains: "Para 3 in the main text".
The paragraph before this one contains: "Para 2 in the main text".
Done
My Computer

Figure 12.3 Selecting the First Preceding <Paragraph> Element.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting a preceding element</title>
</head>
<body>
<h3>This example selects a preceding &lt;Paragraph&gt; element. </h3>
<xsl:apply-templates select="/Manual/MainText/Paragraph[3]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<p>This paragraph contains: "<xsl:value-of select='.'/>".</p>
<p>The paragraph before this one contains: "<xsl:value-of
  select='preceding::Paragraph'/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.13 A Stylesheet to Select a Preceding Named Element (Manual02.xsl).

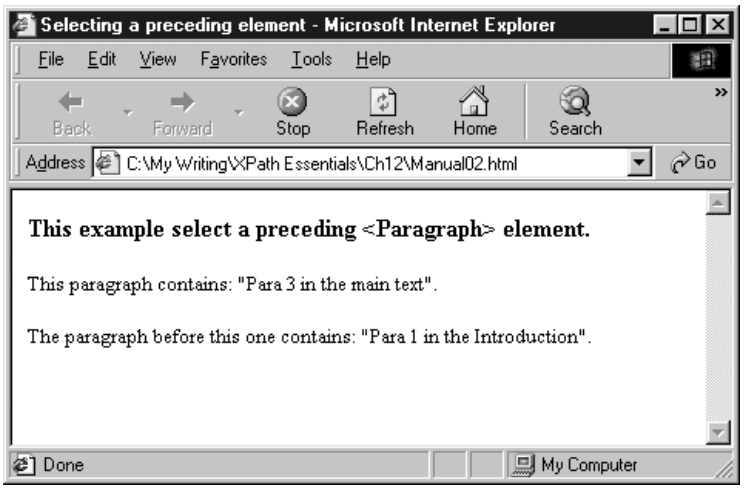

Selecting a preceding element - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch12\Manual02.html
Go
This example select a preceding <Paragraph> element.
This paragraph contains: "Para 3 in the main text".
The paragraph before this one contains: "Para 1 in the Introduction".
Done
My Computer

Figure 12.4 Selecting the Preceding <Paragraph> Elements.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting a preceding sibling element</title>
</head>
<body>
<h3>This example select a preceding sibling &lt;Paragraph&gt; element.
  </h3>
<xsl:apply-templates select="/Manual/Introduction/Paragraph[4]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<p>This paragraph contains: "<xsl:value-of select='.'/>".</p>
<p>The second preceding sibling paragraph before this one contains:
"<xsl:value-of select='preceding-sibling::Paragraph[2]'/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.14 A Stylesheet to Select a Preceding Sibling Element (Manual03.xsl).

The preceding-sibling axis is a reverse axis; therefore, the first Paragraph element node in that axis is the third Paragraph element node and the second (counting backwards) in the preceding-sibling axis is the second Paragraph element node (in document order). The output of the transformation is shown in Figure 12.4.

Selecting Following Elements

You might want to select elements that occur later in document order. We will use Listing 12.15 as our XML source document for transformation.

Notice that on this occasion the <xsl:apply-templates> element in the main template means that the <xsl:template> that matches on Paragraph element nodes will instantiate with the context node as the third Paragraph element node in the introduction.

```
<xsl:apply-templates select="/Manual/Introduction/Paragraph[3]"/>
```

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting a following Paragraph element</title>
</head>
<body>
<h3>This example select a following &lt;Paragraph&gt; element. </h3>
<xsl:apply-templates select="/Manual/Introduction/Paragraph[3]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<p>This paragraph contains: "<xsl:value-of select='.'/>".</p>
<p>The paragraph following this one contains: "<xsl:value-of
  select='following::Paragraph[2]'/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.15 A Stylesheet to Select Elements Occurring Later in Document Order (Manual04.xsl).

We then, within the instantiated template, choose to display the second Paragraph element node in the following axis:

```
<xsl:value-of select='following::Paragraph[2]'/>
```

As you can see in Figure 12.5, this is the first Paragraph element node within the main text.

You might want to more specifically select following elements that are also siblings of the context element. You can do that using XPath's following-sibling axis as demonstrated in Listing 12.16.

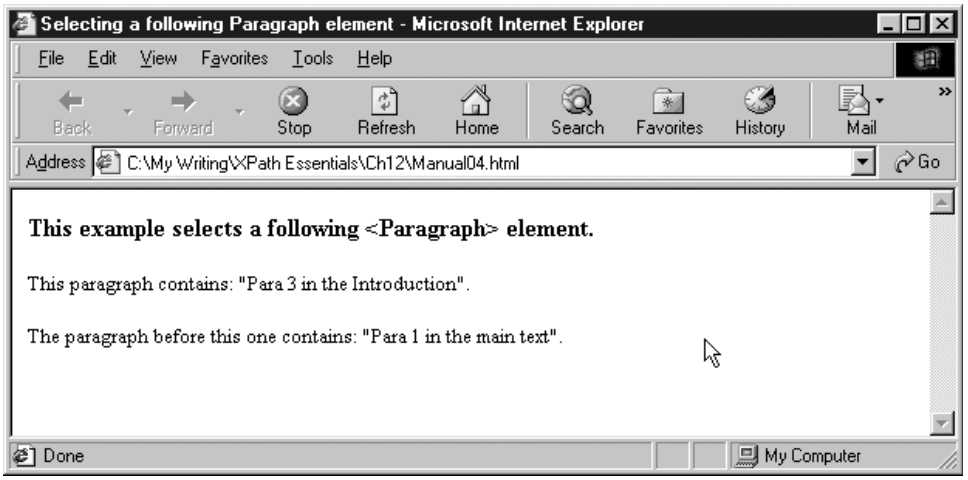

Selecting a following Paragraph element - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\Manual04.html
Go
This example selects a following <Paragraph> element.
This paragraph contains: "Para 3 in the Introduction".
The paragraph before this one contains: "Para 1 in the main text".
Done
My Computer

Figure 12.5 Using the Following Axis.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting a following Paragraph element</title>
</head>
```

Listing 12.16 A Stylesheet to Select a Following Sibling (Manual05.xsl).

```
<body>
<h3>This example selects a following &lt;Paragraph&gt; element. </h3>
<xsl:apply-templates select="/Manual/MainText/Paragraph[1]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<p>This paragraph contains: "<xsl:value-of select='.'/>".</p>
<p>The third paragraph following this one contains: "<xsl:value-of
  select='following::Paragraph[3]'/>".</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.16 (*Continued*)

The <xsl:apply-templates> element in the main template causes an <xsl:template> that matches on Paragraph to instantiate only on the third Paragraph element node within the main text.

```
<xsl:apply-templates select="/Manual/MainText/Paragraph[1]"/>
```

Within the template of the <xsl:value-of> element, this code causes the third Paragraph element node in the following axis to be displayed.

```
<xsl:value-of select='following::Paragraph[3]'/>
```

If you wanted a more specific output in the following-sibling axis, the <xsl:value-of> element could be modified to

```
<xsl:value-of select='following-sibling::Paragraph[3]'/>
```

In this case, the element node selected for display would be the same. The output from Listing 12.16 is shown in Figure 12.6.

Selecting Elements by Attribute Presence

A common task is to select elements either if they have a particular attribute or if they lack a particular attribute. We will use the example of an offering of training courses as illustrated by Listing 12.17. Let's suppose XMML.com offers a range of training courses but wants to display those on special pricing, as indicated by the presence of a "special" attribute.

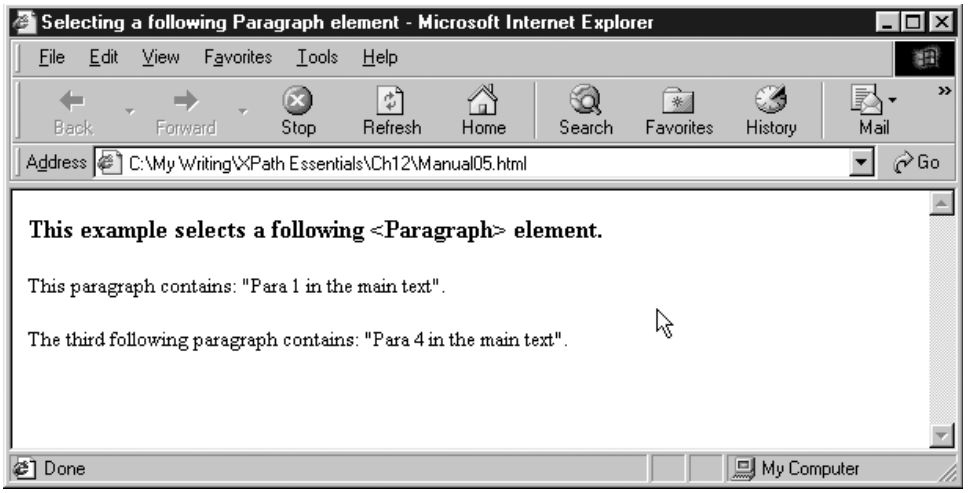

Selecting a following Paragraph element - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\Manual05.html
Go
This example selects a following <Paragraph> element.
This paragraph contains: "Para 1 in the main text".
The third following paragraph contains: "Para 4 in the main text".
Done
My Computer

Figure 12.6 The Output from the Stylesheet in Listing 12.16.

```
<?xml version='1.0'?>
<TrainingCourses>
<Course special="true">
Introduction to XML for HTML authors
</Course>
<Course>
XPath for beginners
</Course>
<Course>
XSLT 101
</Course>
<Course special="true">
XHTML versions 1.0 and 1.1
</Course>
</TrainingCourses>
```

Listing 12.17 A Simple Catalog of Training Courses (TrainingCourses.xml).

We can select <Course> elements that possess a "special" attribute by adding a predicate to the value of the select attribute of the <xsl:apply-templates> element in the main template, as in Listing 12.18.

The predicate [@special] in the select attribute of the <xsl:apply-templates> element indicates that the selected element must possess a "special" attribute.

The resulting HTML is shown in Figure 12.7.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements which possess a particular attribute</title>
</head>
<body>
<h3>Only &lt;Course&gt; elements which possess a "special" attribute
  will be selected and displayed.</h3>
<xsl:apply-templates select="/TrainingCourses/Course[@special]">
<xsl:sort select="."/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Course">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.18 A Stylesheet to Select Elements That Possess a Particular Named Attribute (TrainingCourses.xsl).

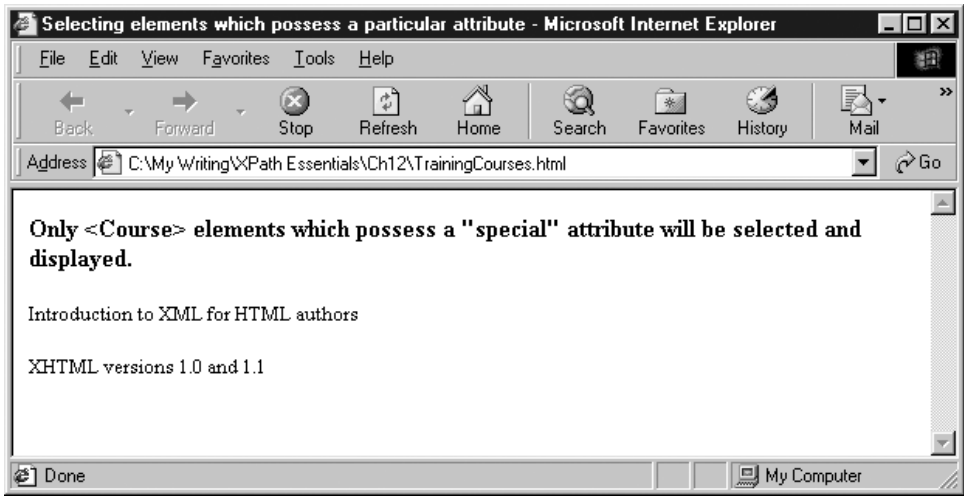

Selecting elements which possess a particular attribute - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\TrainingCourses.html
Go
Only <Course> elements which possess a "special" attribute will be selected and displayed.
Introduction to XML for HTML authors
XHTML versions 1.0 and 1.1
Done
My Computer

Figure 12.7 Selecting Elements That Possess a Particular Attribute.

In some other situation you might want to select elements only if they lack a particular attribute. Listing 12.19 shows a stylesheet that will do that, using Listing 12.17 as the XML source document.

The reversal of the logic of the selection is achieved by the XPath not ( ) function in the <xsl:apply-templates> element in the main template.

```
<xsl:apply-templates select="/TrainingCourses/Course[not(@special)]">
```

The output from Listing 12.19 is shown in Figure 12.8.

You might even want to select elements only if they possessed no attributes at all. For example, in Listing 12.20, you might want to be able to automatically identify elements where the process of adding attributes had been omitted.

The stylesheet in Listing 12.21 selects for display only those <Course> elements that lack any attribute.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements which possess a particular attribute</title>
</head>
<body>
<h3>Only &lt;Course&gt; elements which do NOT possess a "special"
  attribute will be
selected and displayed.</h3>
<xsl:apply-templates select="/TrainingCourses/Course[not(@special)]">
<xsl:sort select="."/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Course">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.19 A Stylesheet to Select Elements That Lack a Particular Named Attribute (TrainingCourses01a.xsl).

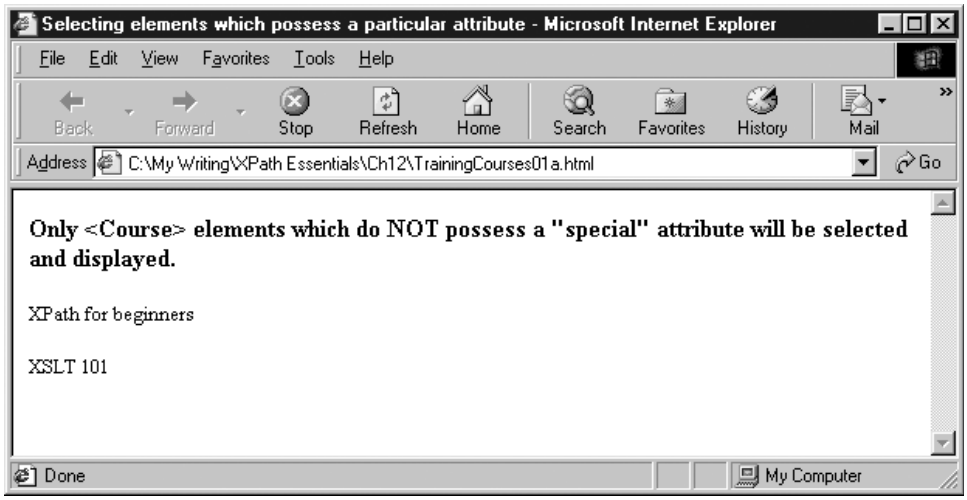

Selecting elements which possess a particular attribute - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\TrainingCourses01a.html
Go
Only <Course> elements which do NOT possess a "special" attribute will be selected and displayed.
XPath for beginners
XSLT 101
Done
My Computer

Figure 12.8 The output from the Stylesheet in Listing 12.19.

```
<?xml version='1.0'?>
<TrainingCourses>
<Course category="XML">
Introduction to XML for HTML authors
</Course>
<Course category="XPath">
XPath for beginners
</Course>
<Course category="SVG">
Introduction to SVG for programmers
</Course>
<Course category="XHTML">
Modularization of XHTML
</Course>
<Course>
Introduction to SVG for designers
</Course>
<Course category="XML">
XSLT 101
</Course>
<Course category="XHTML">
XHTML versions 1.0 and 1.1
</Course>
<Course>
SVG Animations
</Course>
</TrainingCourses>
```

Listing 12.20 A Catalog of Training Courses, in Which Some Elements Have Attributes (TrainingCourses02.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements which lack  attributes.</title>
</head>
<body>
<h3>Only &lt;Course&gt; elements which have no attributes will be
  selected and displayed.</h3>
<xsl:apply-templates select="/TrainingCourses/Course[not(@*)]">
<xsl:sort select="."/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Course">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

TEAMFLY

Listing 12.21 A Stylesheet to Display Elements That Lack Attributes (TrainingCourses02.xsl).

The important part of the stylesheet is the <xsl:apply-templates> element in the main template, where the predicate `[not(@*)]` selects only elements where there are no attributes present.

```
<xsl:apply-templates select="/TrainingCourses/Course[not(@*)]">
```

Figure 12.9 shows the output after applying Listing 12.21 to Listing 12.20.

Selecting Elements by Attribute Value

It is often the case that you may want to select an element, or elements, if they possess a particular attribute and that attribute has a particular value. Listing 12.22 provides a source document to allow us to select elements on that basis.

Listing 12.23 shows a stylesheet that selects for display only those <Course> elements that possess a category attribute with the value of "XML".

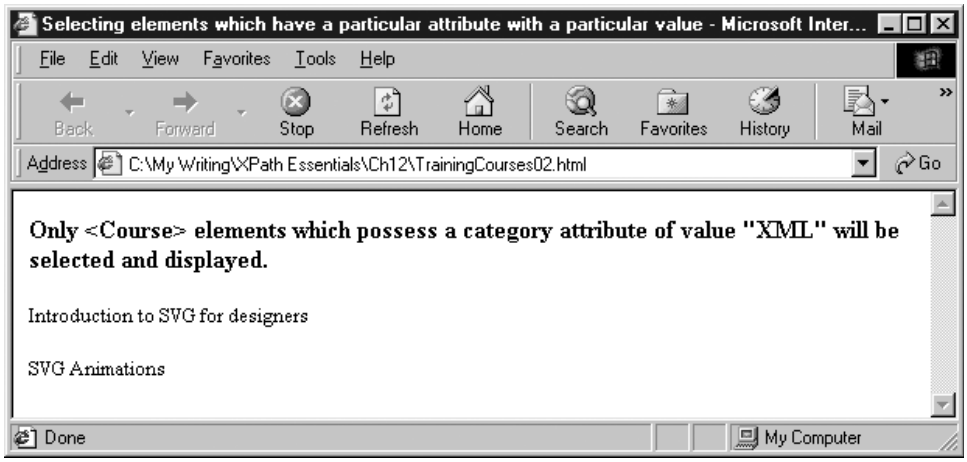

Selecting elements which have a particular attribute with a particular value - Microsoft Inter...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\TrainingCourses02.html
Go
Only <Course> elements which possess a category attribute of value "XML" will be selected and displayed.
Introduction to SVG for designers
SVG Animations
Done
My Computer

Figure 12.9 Selectively Displaying Only <Course> Elements That Lack Any Attribute.

```
<?xml version='1.0'?>
<TrainingCourses>
<Course category="XML">
Introduction to XML for HTML authors
</Course>
<Course category="XPath">
XPath for beginners
</Course>
<Course category="SVG">
Introduction to SVG for programmers
</Course>
<Course category="XHTML">
Modularization of XHTML
</Course>
<Course category="XML">
XSLT 101
</Course>
<Course category="XHTML">
XHTML versions 1.0 and 1.1
</Course>
</TrainingCourses>
```

Listing 12.22 A Simple Catalog of Training Courses (TrainingCourses03.xml) .

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements which have a particular attribute with a
  particular value</title>
</head>
<body>
<h3>Only &lt;Course&gt; elements which possess a category attribute of
  value "XML" will be selected and displayed.</h3>
<xsl:apply-templates select="/TrainingCourses/Course[@category='XML']">
<xsl:sort select="."/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Course">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.23 A Stylesheet to Select Elements That Possess a Particular Attribute Having a Particular Value (TrainingCourses03.xsl).

The <xsl:apply-templates> element has a predicate within its value attribute that specifies that the category attribute must be present and must have a value of "XML".

```
<xsl:apply-templates select="/TrainingCourses/Course[@category='XML']">
```

The output, which shows only <Course> elements with a category attribute of value "XML", is shown in Figure 12.10.

This technique can, of course, be extended to select only elements that have two attributes, each of some particular value, by adding a further predicate to the location path. For example, if (with a slightly different source document) we wanted to select only <Course> elements that had a category element of value "XML" and a price attribute of value "Special", we could use the following location path:

```
/TrainingCourses/Course[@category="XML"][@price="Special"]
```

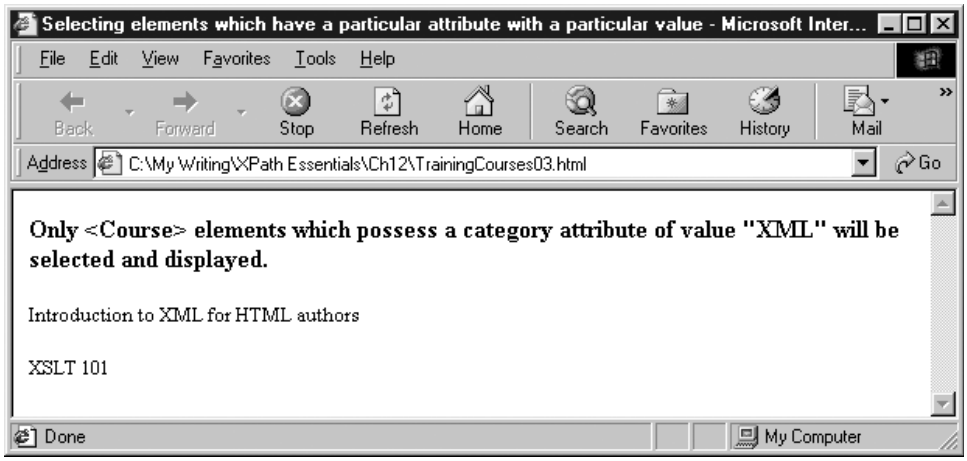

Selecting elements which have a particular attribute with a particular value - Microsoft Inter...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\TrainingCourses03.html
Go
Only <Course> elements which possess a category attribute of value "XML" will be selected and displayed.
Introduction to XML for HTML authors
XSLT 101
Done
My Computer

Figure 12.10 Selecting Elements by the Presence of an Attribute with a Particular Value.

Attribute Less Than a Specified Value

Let's suppose that we want to select for display elements that have a price less than 30 (whatever the currency units may be).

Our XML source document is shown in Listing 12.24, with price information stored in the price attribute on the <Item> element.

We can select for display the elements with a price less than 30 by embedding an <xsl:if> element within the template, which matches the <Item> element and compares the value of the price attribute to 30, as in the stylesheet in Listing 12.25.

Notice that within the test attribute of the <xsl:if> element, the less than sign is escaped and replaced by "<".

```
<xsl:if test="@price&lt;30">
```

If that isn't done, an error will occur when you attempt to process the stylesheet.

An alternative approach is to use a predicate within the value of the select attribute in the <xsl:apply-templates> element, as in Listing 12.26.

```
<?xml version='1.0'?>
<Items>
<Item price="50">Widget</Item>
<Item price="30">Thing</Item>
<Item price="28">Wotsit</Item>
<Item price="43">OneOfThose</Item>
<Item price="51">ExpensiveThing</Item>
</Items>
```

Listing 12.24 A Simple Price List Expressed in XML (Items.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Selecting elements on the value of an attribute</title>
</head>
<body>
<xsl:apply-templates select="Items/Item"/>
</body>
</html>
</xsl:template>

<xsl:template match="Item">
<xsl:if test="@price&lt;30">
<p><xsl:value-of select="."/></p>
</xsl:if>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.25 A Stylesheet to Select Elements on the Basis of Attribute Value Less Than a Selected Value (Items.xsl).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Selecting elements on the value of an attribute</title>
</head>
<body>
<xsl:apply-templates select="Items/Item[@price&lt;30]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Item">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 12.26 A Stylesheet Showing an Alternative Approach to Selecting an Element with an Attribute Less Than a Selected Value (Items2.xsl).

Notice that the predicate selects for application only those <Item> elements where the value of the price attribute is less than 30, and, therefore, there is no need for any test to be contained within the <xsl:template> element which matches an <Item> element, since the select attribute of the <xsl:apply-templates> element ensures that only element nodes whose price value is less than 30 are processed.

Selecting Elements When Passing Parameters

In an earlier example I showed you how to select elements on the basis of the content of the element. However, in practice you may want to have something that is much more flexible and which allows you to determine at runtime what element or elements are to be selected for transformation.

To enable you to do that, XSLT provides the <xsl:param> element. The <xsl:param> element exists as a so-called top-level element in an XSLT stylesheet.

It is likely that your favorite XSLT processor will accept parameters on the command line. Here I will describe how to pass parameters to a stylesheet when using Instant Saxon.

Listing 12.27 is a very simple XML-based team diary, which will be sufficient to allow us to explore how to handle parameters passed on the command line, and, on the basis of those parameters, to select elements for display.

The stylesheet in Listing 12.28 will allow us to display the activities scheduled for a certain date, based on the parameters passed on the command line.

```
<?xml version='1.0'?>
<TeamDiary>
<Person name="Andrew">
<Date>
<Month>March</Month>
<Day>21</Day>
<Slot>PM</Slot>
</Date>
<Activity>Client Visit</Activity>
</Person>
<Person name="Jenny">
<Date>
<Month>March</Month>
<Day>21</Day>
<Slot>AM</Slot>
</Date>
<Activity>Planning Meeting</Activity>
</Person>
<Person name="Andrew">
```

Listing 12.27 A Simple Team Diary in XML (TeamDiary.xml).

(*continues*)

```
<Date>
<Month>March</Month>
<Day>22</Day>
<Slot>AM</Slot>
</Date>
<Activity>Business Development</Activity>
</Person>
<Person name="Jonathan">
<Date>
<Month>March</Month>
<Day>22</Day>
<Slot>PM</Slot>
</Date>
<Activity>Tamir Project</Activity>
</Person>
<Person name="Andrew">
<Date>
<Month>March</Month>
<Day>23</Day>
<Slot>AM</Slot>
</Date>
<Activity>Management Seminar</Activity>
</Person>
<Person name="Jenny">
<Date>
<Month>March</Month>
<Day>22</Day>
<Slot>AM</Slot>
</Date>
<Activity>Design Meeting</Activity>
</Person>
</TeamDiary>
```

Listing 12.27 (*Continued*)

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:param name="Month"/>
<xsl:param name="Day"/>

<xsl:template match="/">
```

Listing 12.28 A Stylesheet to Select Events Taking Place on a Certain Date (TeamDiary.xsl).

```
<html>
<head>
<title>Using &lt;xsl:param&gt; to select elements</title>
</head>
<body>
<h3>Selecting people's activities by the date passed to the
  stylesheet.</h3>
<p>The month passed as a parameter was: <xsl:value-of
  select="$Month"/></p>
<p>The day passed as a parameter was: <xsl:value-of select="$Day"/></p>
<h3>The following activities are scheduled for <xsl:value-of
  select="$Month"/> <xsl:value-of select="$Day"/>.</h3>
<xsl:apply-templates
select="TeamDiary/Person/Date[Month=$Month][Day=$Day]">
<xsl:sort select="Slot"/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="Date">
<p>Slot:<xsl:value-of select="Slot"/><br />
Person:<xsl:value-of select="../@name"/><br />
Activity:<xsl:value-of select="../Activity"/></p>

</xsl:template>
</xsl:stylesheet>
```

Listing 12.28 (*Continued*)

NOTE When you are using predicates that involve a comparison of the value of an element with the variable created as a result of passing the parameter, do not include quotation marks around $Variable. If you do, you will probably not get the hoped-for output.

If on the command line we enter

```
Saxon TeamDiary.xml TeamDiary.xsl Month=March Day=21 > TeamDiary.html
```

we are telling Instant Saxon to apply the stylesheet TeamDiary.xsl to the source document TeamDiary.xml using the passed parameters of

```
Month=March
```

and

```
Day=21
```

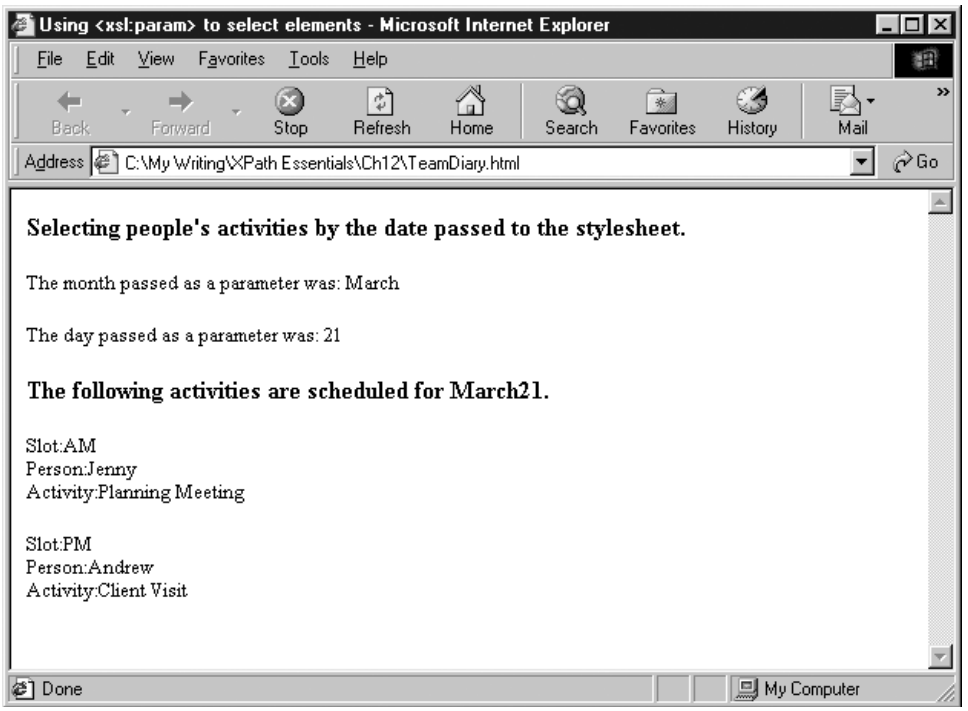

Using <xsl:param> to select elements - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\TeamDiary.html
Go
Selecting people's activities by the date passed to the stylesheet.
The month passed as a parameter was: March
The day passed as a parameter was: 21
The following activities are scheduled for March21.
Slot:AM
Person:Jenny
Activity:Planning Meeting
Slot:PM
Person:Andrew
Activity:Client Visit
Done
My Computer

Figure 12.11 A Demonstration of Passing Parameters and Using Them to Select Elements for Display.

Notice that quotes are not used in the command line.

Within the stylesheet the parameters are, by means of the <xsl:param> top-level element, converted into global variables that we can employ in location paths within the stylesheet.

With the command line noted above, we obtain the output shown in Figure 12.11.

It is straightforward to refine the output a little to produce an HTML table to display the selected content from the source document, which is output. Listing 12.29 adapts the stylesheet to achieve that.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:param name="Month"/>
<xsl:param name="Day"/>

<xsl:template match="/">
```

Listing 12.29 A Stylesheet to Create an HTML Table Displaying Diary Entries for a Selected Date (TeamDiary2.xsl).

```
<html>
<head>
<title>Using &lt;xsl:param&gt; to select elements</title>
</head>
<body>
<h3>Selecting people's activities by the date passed to the
stylesheet.</h3>
<p>The month passed as a parameter (the variable $Month) was:
  <xsl:value-of select="$Month"/></p>
<p>The day passed as a parameter (the variable $Day) was: <xsl:value-of
  select="$Day"/></p>
<h3>The following activities are scheduled for <xsl:value-of
  select="$Month"/><xsl:text> </xsl:text><xsl:value-of
  select="$Day"/>.</h3>
<table cellspacing="2" cellpadding="3" border="3">
<tr><td> Slot (AM/PM)</td>
<td>Person</td>
<td>Activity</td>
</tr>
<xsl:apply-templates
select="TeamDiary/Person/Date[Month=$Month][Day=$Day]">
<xsl:sort select="Slot"/>
</xsl:apply-templates>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="Date">
<tr>
<td><xsl:value-of select="Slot"/></td>
<td><xsl:value-of select="../@name"/></td>
<td><xsl:value-of select="../Activity"/></td>
</tr>

</xsl:template>
</xsl:stylesheet>
```

Listing 12.29 (*Continued*)

Within the stylesheet, the following code defines the names of the parameters that will be processed within the stylesheet.

```
<xsl:param name="Month"/>
<xsl:param name="Day"/>
```

These appear in the <xsl:apply-templates> element in the main template.

```
<xsl:apply-templates select="TeamDiary/Person/Date[Month=$Month]
  [Day=$Day]">
```

The two predicates indicate that the <Date> elements selected have a Month element node child equal to the parameter $Month, which was passed into the stylesheet, and, additionally, that the Day child element node equals the $Day parameter. See Figure 12.12.

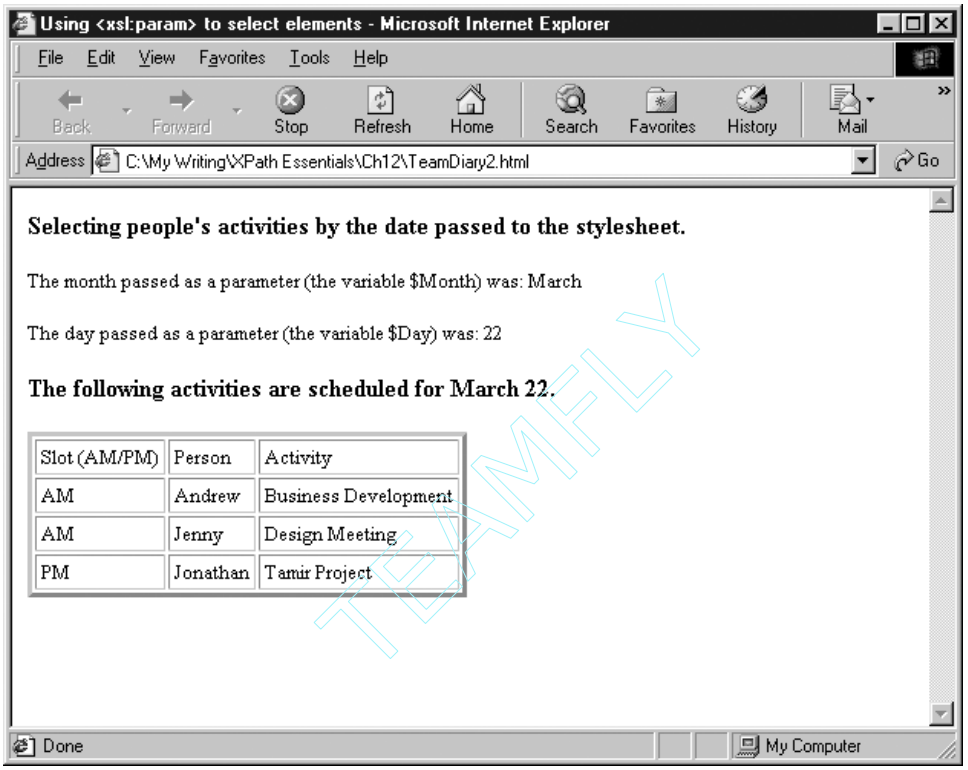
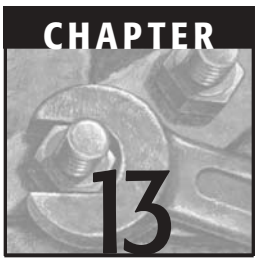

Using <xsl:param> to select elements - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch12\TeamDiary2.html
Go
Selecting people's activities by the date passed to the stylesheet.
The month passed as a parameter (the variable $Month) was: March
The day passed as a parameter (the variable $Day) was: 22
The following activities are scheduled for March 22.
Slot (AM/PM) Person Activity
AM Andrew Business Development
AM Jenny Design Meeting
PM Jonathan Tamir Project
Done
My Computer

Figure 12.12 Outputting an HTML Table Determined by a Parameter Passed to the Stylesheet.

Looking Ahead

Chapter 13 provides further examples of the use of XPath functions.



Working with XPath Functions

This chapter will further demonstrate the use of the XPath functions that you were introduced to in Chapter 5. For those who have worked through the book, this should provide reinforcement and further examples of ideas that have already been discussed.

Using Node-Set Functions

In this section we will look at some examples of using the XPath node-set functions.

Using the count( ) Function

A possible use for the count ( ) function is to count the number of elements of a particular type. Let's suppose we have a database of purchase orders held as XML and we want to find out, perhaps for the purpose of targeting customers who buy more than a certain amount, which purchase orders involved the purchase of four or more line items.

Our source XML might look like that in Listing 13.1.

The stylesheet in Listing 13.2 uses the count ( ) function as part of the expression that forms the value of the select attribute of the <xsl:apply-templates> element. Therefore, only those <PurchaseOrder> elements that have four or more <LineItem> element children are processed using the <xsl:template> element with match attribute having a value of "PurchaseOrder".

```
<?xml version='1.0'?>
<PurchaseOrders>
<PurchaseOrder number="0010" customerID="1234">
<LineItem quantity="">Some item</LineItem>
<LineItem quantity="">Some other item</LineItem>
<LineItem quantity="">Something different</LineItem>
<LineItem quantity="">Something Else Again</LineItem>
</PurchaseOrder>
<PurchaseOrder number="0011" customerID="2345">
<LineItem quantity="">A thing</LineItem>
<LineItem quantity="">Another thing</LineItem>
<LineItem quantity="">A different thing</LineItem>
</PurchaseOrder>
<PurchaseOrder number="0012" customerID="3456">
<LineItem quantity="">An item</LineItem>
<LineItem quantity="">Widget</LineItem>
<LineItem quantity="">DooDaa</LineItem>
<LineItem quantity="">Wotsit</LineItem>
</PurchaseOrder>
<PurchaseOrder number="0013" customerID="4567">
<LineItem quantity="">ice cream</LineItem>
<LineItem quantity="">mousse</LineItem>
<LineItem quantity="">peanuts</LineItem>
<LineItem quantity="">diamond</LineItem>
<LineItem quantity="">watch</LineItem>
</PurchaseOrder>
</PurchaseOrders>
```

Listing 13.1 A Brief List of Purchase Orders Expressed in XML (PurchaseOrders.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Purchases of four or more items</title>
</head>
<body>
<h3>XMML.com - Customer purchasing survey.</h3>
<p>The following purchases were of four items or more.</p>
<br />
<xsl:apply-templates
```

Listing 13.2 A Stylesheet to Demonstrate the count ( ) Function (PurchaseOrders.xsl).

```
select="/PurchaseOrders/PurchaseOrder[count(LineItem)>3]"/>

</body>
</html>
</xsl:template>

<xsl:template match="PurchaseOrder">
<p>Order Number: <xsl:value-of select="@number"/><br />
Customer ID: <xsl:value-of select="@customerID"/></p>
<br />

</xsl:template>

</xsl:stylesheet>
```

Listing 13.2 (*Continued*)

The output from applying the XSLT stylesheet in Listing 13.2 to the source XML in Listing 13.1 is shown in Figure 13.1.

In the next example we will use the count ( ) function to display the number of cities from each of two countries when we display the cities in an output document. The source XML document is shown in Listing 13.3.

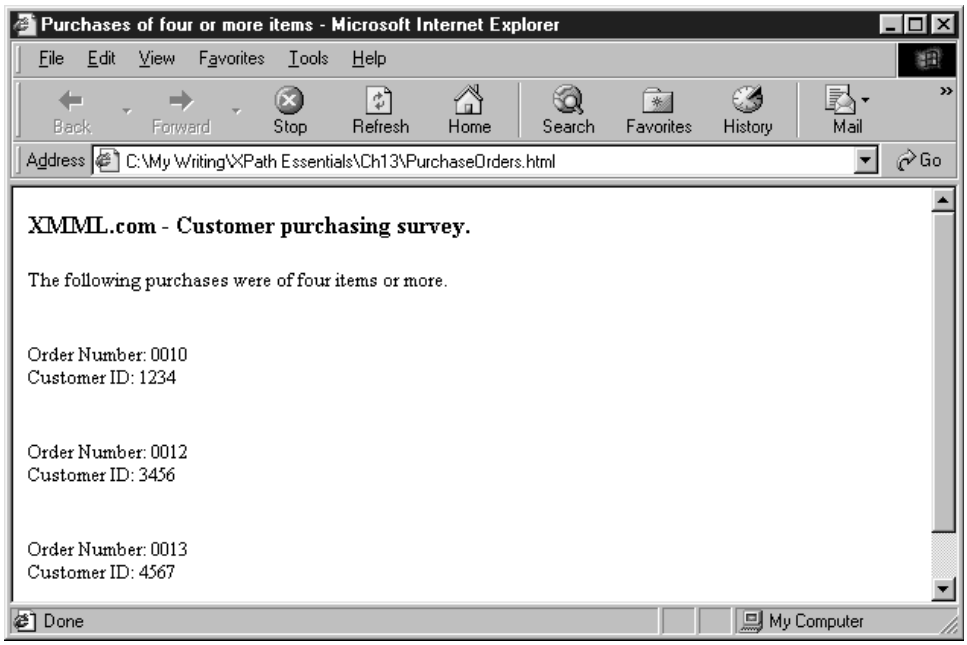

Purchases of four or more items - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\PurchaseOrders.html
Go
XMML.com - Customer purchasing survey.
The following purchases were of four items or more.
Order Number: 0010
Customer ID: 1234
Order Number: 0012
Customer ID: 3456
Order Number: 0013
Customer ID: 4567
Done
My Computer

Figure 13.1 Use of the count ( ) Function to Select Purchase Orders That Consist of Four or More Line Items.

```
<?xml version='1.0'?>
<Cities>
<City country="USA">New York</City>
<City country="USA">San Francisco</City>
<City country="UK">Birmingham</City>
<City country="UK">Aberdeen</City>
<City country="USA">Denver</City>
<City country="USA">Tampa</City>
<City country="UK">London</City>
<City country="UK">Edinburgh</City>
<City country="USA">Miami</City>
<City country="USA">Boston</City>
<City country="UK">Manchester</City>
<City country="USA">Seattle</City>
<City country="UK">Glasgow</City>
<City country="USA">Chicago</City>
</Cities>
```

Listing 13.3 A List of Cities Expressed in XML (Cities.xml).

The XSLT stylesheet in Listing 13.4 counts the number of cities from each country that are mentioned and outputs that count when the list of cities in that country is displayed.

The output is shown in Figure 13.2.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>List of cities in the USA and the UK</title>
</head>
<body>
<h3>A list of <xsl:value-of select="count(/Cities/City[@country=
  'USA'])"/> cities in the USA</h3>
<xsl:apply-templates select="/Cities/City[@country='USA']">
```

Listing 13.4 A Stylesheet to Use the count ( ) Function to Count Cities from Each of Two Countries (Cities.xsl).

```
<xsl:sort select="."/>
</xsl:apply-templates>
<br />
<h3>A list of <xsl:value-of
select="count(/Cities/City[@country='UK'])"/> cities in the UK</h3>
<xsl:apply-templates select="/Cities/City[@country='UK']">
<xsl:sort select="."/>
</xsl:apply-templates>
</body>
</html>
</xsl:template>

<xsl:template match="City">
<xsl:value-of select="."/><br />
</xsl:template>

</xsl:stylesheet>
```

Listing 13.4 (*Continued*)

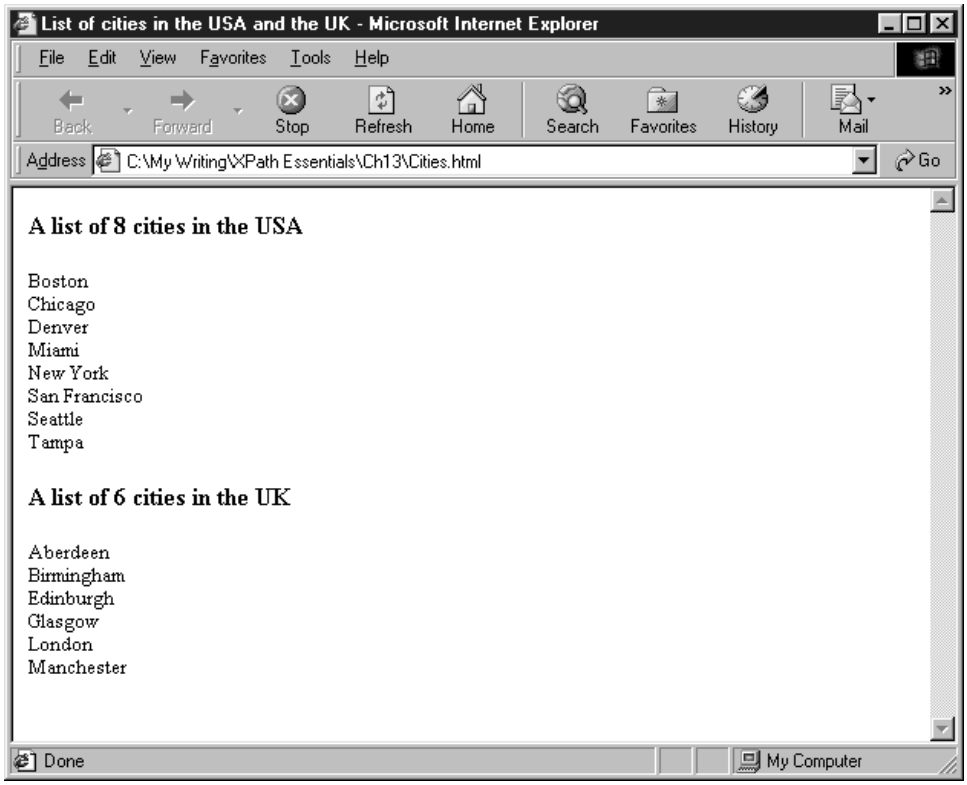


Figure 13.2 Using the count() Function to Display Results.

Using the id( ) Function

The id ( ) function allows us to select an element on the basis of an ID attribute. You may recall that an ID attribute must be declared as such in a Document Type Definition (DTD).

To be able to use the id ( ) function, an attribute must be declared as of type ID in a DTD that accompanies an XML source document. First, let's create a source document, shown in Listing 13.5.

```
<?xml version='1.0'?>
<!DOCTYPE PurchaseOrders SYSTEM "PurchaseOrdersID.dtd">
<PurchaseOrders>
<PurchaseOrder id="ABC123">
<Customer>John Smith &amp;amp; Co.</Customer>
<LineItems>
<LineItem qty="4">Copy paper</LineItem>
<LineItem qty="3">Highlight pens</LineItem>
<LineItem qty="5">Pencils</LineItem>
<LineItem qty="12">Felt tip pens</LineItem>
<LineItem qty="1">Stapler</LineItem>
</LineItems>
</PurchaseOrder>
<PurchaseOrder id="ABC124">
<Customer>Grampian Online Design</Customer>
<LineItems>
<LineItem qty="1">Mouse mat</LineItem>
<LineItem qty="1">Natural keyboard</LineItem>
<LineItem qty="2">Desk Clock</LineItem>
<LineItem qty="4">Red felt pens</LineItem>
<LineItem qty="2">Black marker pens</LineItem>
</LineItems>
</PurchaseOrder>
<PurchaseOrder id="ABC125">
<Customer>John Smith &amp;amp; Co.</Customer>
<LineItems>
<LineItem qty="1">Standard keyboard</LineItem>
<LineItem qty="1">External modem</LineItem>
<LineItem qty="30">3.5" Floppy Disks</LineItem>
<LineItem qty="20">CD Rewritable</LineItem>
<LineItem qty="4">Pencils</LineItem>
</LineItems>
</PurchaseOrder>
<PurchaseOrder id="ABC126">
<Customer>Paw's K9 Supplies</Customer>
<LineItems>
```

Listing 13.5 A List of Purchase Orders, Each of Which Has an ID Attribute (PurchaseOrdersID.xml).

```
<LineItem qty="48">Yellow sticky notes, small</LineItem>
<LineItem qty="1">Paper clips, box</LineItem>
<LineItem qty="500">Envelopes, small</LineItem>
<LineItem qty="250">Envelopes, large</LineItem>
<LineItem qty="3">Highlight pens</LineItem>
</LineItems>
</PurchaseOrder>
</PurchaseOrders>
```

Listing 13.5 (*Continued*)

Next we need to have a Document Type Definition, which corresponds to the XML source document. This is shown in Listing 13.6.

The XSLT stylesheet, which will display a purchase order on the basis of the value returned by the id ( ) function, is shown in Listing 13.7.

```
<!ELEMENT PurchaseOrders (PurchaseOrder)*>
<!ELEMENT PurchaseOrder (Customer, LineItems)>
<!ATTLIST PurchaseOrder
  id ID #IMPLIED>
<!ELEMENT Customer (#PCDATA)>
<!ELEMENT LineItems (LineItem*)>
<!ELEMENT LineItem (#PCDATA)>
<!ATTLIST LineItem
  qty CDATA #IMPLIED>
```

Listing 13.6 The Document Type Definition (DTD) for Listing 13.5 (PurchaseOrdersID.dtd).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using the id() function</title>
```

Listing 13.7 A Stylesheet to Demonstrate the id ( ) Function (PurchaseOrdersID.xsl).
(*continues*)

```
</head>
<body>
<h3>This example uses the id() function to output a selected data
  item.</h3>
<xsl:apply-templates select="id('ABC124')"/>
</body>
</html>
</xsl:template>

<xsl:template match="id('ABC124')">
<p>PO No.: <xsl:value-of select="@id"/><br />
<xsl:for-each select="LineItems/LineItem">
<xsl:value-of select="."/> - <xsl:value-of select="@qty"/><br />
</xsl:for-each>
</p>
</xsl:template>

</xsl:stylesheet>
```

Listing 13.7 (*Continued*)

As you can see, the id ( ) function is used both within the select attribute of the <xsl:apply-templates> element in the main template and also as the value of the match attribute for the <xsl:template> element that is instantiated.

The output of the transformation is shown in Figure 13.3.

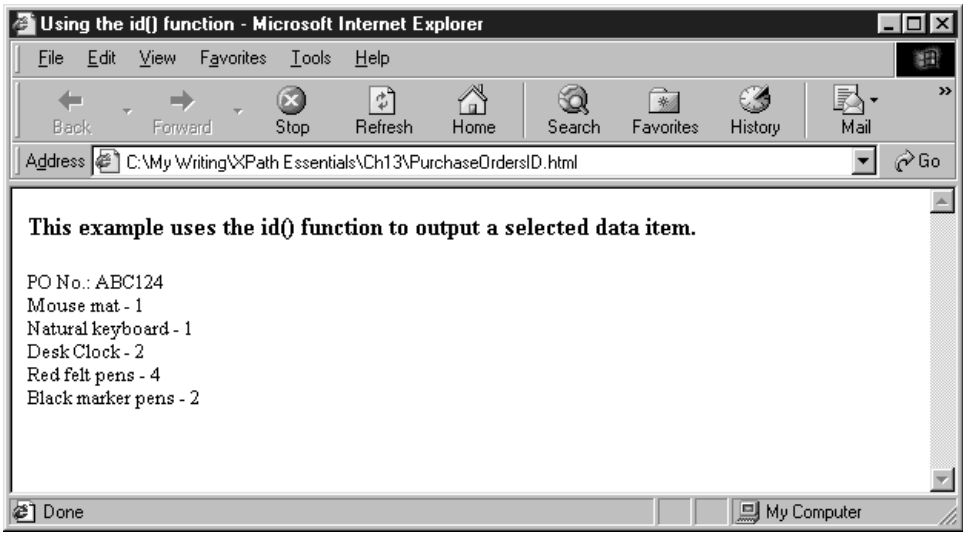

Using the id() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\PurchaseOrdersID.html
Go
This example uses the id() function to output a selected data item.
PO No.: ABC124
Mouse mat - 1
Natural keyboard - 1
Desk Clock - 2
Red felt pens - 4
Black marker pens - 2
Done
My Computer

Figure 13.3 Applying the id ( ) Function.

Using the last( ) Function

Let's suppose we have a report that is updated at irregular intervals, yet it is important that when we access the report, we see the very latest version. Since the report may have been updated several times or may not have been updated at all since we last accessed it, we can't use a date or number in a series, since it isn't easily predictable what that might be. The XPath last ( ) function provides the ideal solution for such a scenario.

Our XML source document looks like Listing 13.8.

If we wanted to access the current safety report at the end of December 2001, we could use the XSLT stylesheet in Listing 13.9, to produce an appropriate output.

```
<?xml version='1.0'?>
<SafetyProcedures>
<SafetyReport>
<Date>March 2001</Date>
<Version>1</Version>
<Text>This is the March 2001 safety report.</Text>
</SafetyReport>
<SafetyReport>
<Date>July 2001</Date>
<Version>2</Version>
<Text>This is the July 2001 safety report.</Text>
</SafetyReport>
<SafetyReport>
<Date>December 2001</Date>
<Version>3</Version>
<Text>This is the December 2001 safety report.</Text>
</SafetyReport>
</SafetyProcedures>
```

Listing 13.8 A Catalog of Irregularly Updated Safety Reports (SafetyReport1.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="html" indent="yes"/>

<xsl:template match="/">
<html>
<head>
<title>XMML.com - Safety Report Version <xsl:value-of select="/
  SafetyProcedures/SafetyReport[last()]/Version"/><xsl:text>
```

Listing 13.9 A Stylesheet to Demonstrate the last ( ) Function (SafetyReport1.xsl).

(*continues*)

```
  </xsl:text>
 <xsl:value-of select="/SafetyProcedures/SafetyReport[last()]/Date"
  /></title>
</head>
<body>
<h1>XMML.com - Safety Report</h1>
<h2>Version <xsl:value-of
select="/SafetyProcedures/SafetyReport[last()]/Version"/> of <xsl:value-
  of select="/SafetyProcedures/SafetyReport[last()]/Date"/></h2>
<p><xsl:value-of select="/SafetyProcedures/SafetyReport[last()]/
  Text"/></p>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.9 (*Continued*)

The last ( ) function was used several times in the XSLT stylesheet, each time as a predicate, `[last()]`, to ensure that the date, version number, or text of the safety report was derived from the latest version, as you can see in Figure 13.4.

The following code includes XPath location paths:

```
<h2>Version <xsl:value-of select="/SafetyProcedures/SafetyReport
  [last()]/Version"/> of <xsl:value-of select="/SafetyProcedures/
  SafetyReport[last()]/Date"/></h2>
```

TEAM-FLY

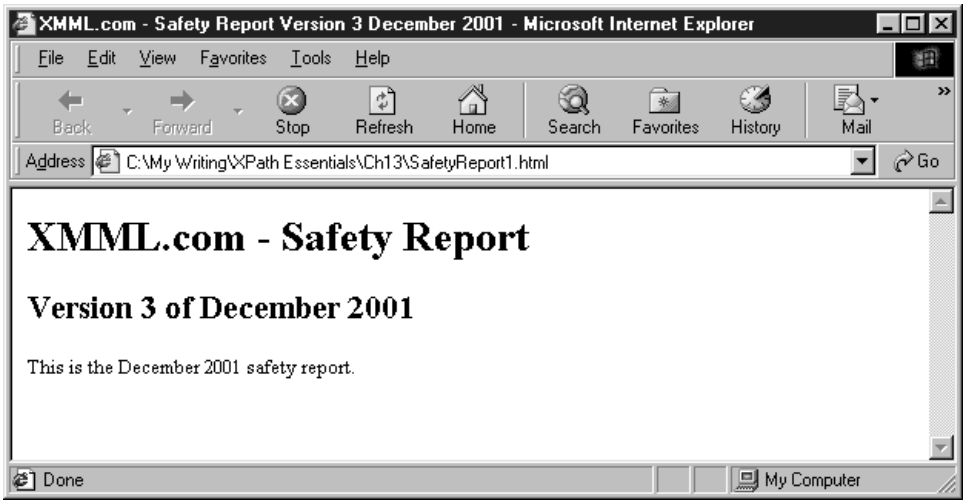

XMML.com - Safety Report Version 3 December 2001 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\SafetyReport1.html
Go
XMML.com - Safety Report
Version 3 of December 2001
This is the December 2001 safety report.
Done
My Computer

Figure 13.4 Selecting the Last Report Using the last ( ) Function.

These location paths indicate that from the root node, look for a <SafetyProcedures> element, then look for any <SafetyReport> elements that it has as children, then choose the last of those. And for that last <SafetyReport> element, display the content of its Version child elements and its Date child elements, respectively.

If we want to run that same report again in March 2002, and it has been updated in both January and February 2002, as in Listing 13.10, we can still use the stylesheet in Listing 13.9 to display the latest version.

The same XSLT stylesheet (Listing 13.9) produces an up-to-date safety report, as you can see in Figure 13.5, without the user having to know whether or not any updates have been made in the interim.

Using the local-name( ) Function

The local-name ( ) function is fairly easy to use. We will use Listing 13.11 as our XML source document.

```
<?xml version='1.0'?>
<SafetyProcedures>
<SafetyReport>
<Date>March 2001</Date>
<Version>1</Version>
<Text>This is the March 2001 safety report.</Text>
</SafetyReport>
<SafetyReport>
<Date>July 2001</Date>
<Version>2</Version>
<Text>This is the July 2001 safety report.</Text>
</SafetyReport>
<SafetyReport>
<Date>December 2001</Date>
<Version>3</Version>
<Text>This is the December 2001 safety report.</Text>
</SafetyReport>
<SafetyReport>
<Date>January 2002</Date>
<Version>4</Version>
<Text>This is the January 2002 safety report.</Text>
</SafetyReport>
<SafetyReport>
<Date>February 2002</Date>
<Version>5</Version>
<Text>This is the February 2002 safety report.</Text>
</SafetyReport>
</SafetyProcedures>
```

Listing 13.10 A Further Version of Listing 13.8, with Further Updates (SafetyReport2.xml).

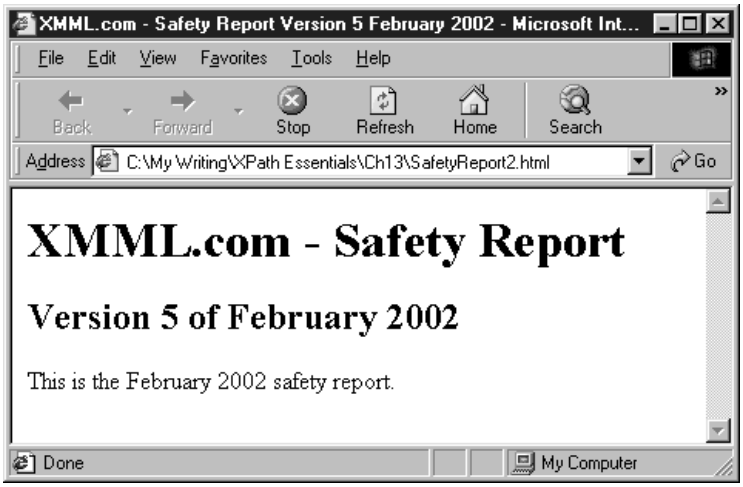

XMML.com - Safety Report Version 5 February 2002 - Microsoft Int...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address C:\My Writing\XPath Essentials\Ch13\SafetyReport2.html
Go
XMML.com - Safety Report
Version 5 of February 2002
This is the February 2002 safety report.
Done
My Computer

Figure 13.5 The Safety Report Displayed Updates Automatically Using the last ( ) Function.

```
<?xml version='1.0'?>
<XMML:Report xmlns:XMML="http://www.XMML.com/Reports/">
<XMML:Title>Report title goes here.</XMML:Title>
<XMML:Author>Fred Jabolowski</XMML:Author>
<XMML:Summary>This summarizes what the report is about.</XMML:Summary>
<XMML:Introduction>Here is an introduction to the
report</XMML:Introduction>
<XMML:MainText>This is the main text of the report.</XMML:MainText>
</XMML:Report>
```

Listing 13.11 A Report Using Namespaces (XMMLReport.xml).

We can use the stylesheet in Listing 13.12 to display the local part for each element in the source document.

NOTE **The part of a QName after the namespace prefix and colon is called the *local part*. Confusingly, the function that selects the local part is called *local-name( ).***

We select all elements in the document to have a template instantiated, using the following code in the main template.

```
<xsl:apply-templates select="//*"/>
```

The <xsl:value-of> elements in the template that matches on "*" simply displays the element name using the name ( ) function and then displays the corresponding local part using the local-name ( ) function.

The output of the transformation is shown in Figure 13.6.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using the local-name() function.</title>
</head>
<body>
<xsl:apply-templates select="//*"/>
</body>
</html>
</xsl:template>

<xsl:template match="*">
<p>The <b>&lt;<xsl:value-of select="name(.)"/>&gt;</b> element has a
  local part <b><xsl:value-of select="local-name(.)"/></b></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.12 A Stylesheet to Demonstrate the local-name ( ) Function (XMMLReport.xsl).

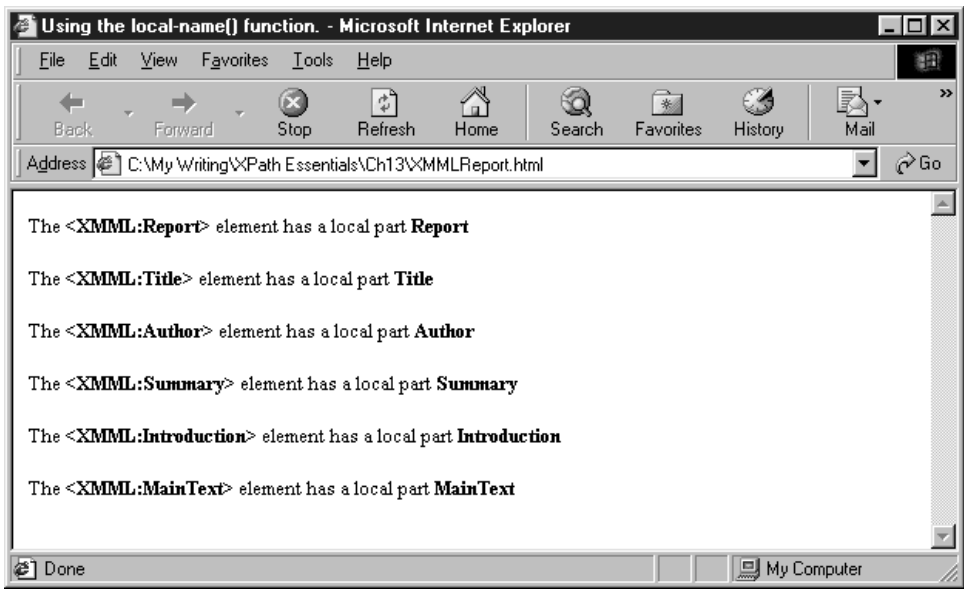


Figure 13.6 Displaying the Local Part of a QName Using the local-name ( ) Function.

Using the name( ) Function

The name ( ) function can be used to find the name of an XPath node. If we had the source document shown in Listing 13.13, we might want to display the name of each of the element child nodes of the <Parent> element.

We could do that using the name ( ) function as in Listing 13.14.

The name ( ) function is used within the <xsl:template> element that matches a Parent element node. The <xsl:for-each> element selects the element node children, sorts

```
<?xml version='1.0'?>
<Parent name="father">
<Daughter name="Martha"/>
<Daughter name="Mary"/>
<Son name="Larry"/>
<Son name="Peter"/>
</Parent>
```

Listing 13.13 A Family Expressed in XML (Parent.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the name() function on element nodes</title>
</head>
<body>
<h3>Using the name() function to display the name of element nodes</h3>
<xsl:apply-templates select="Parent"/>

</body>
</html>
</xsl:template>

<xsl:template match="Parent">
<xsl:for-each select="*">
<xsl:sort select="."/>
<p>The name of the element node in position <xsl:value-of select=
  "position()"/> is <xsl:value-of select="name()"/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.14 A Stylesheet to Demonstrate the name ( ) Function (Parent.xsl).

them by content (which does nothing since the corresponding elements happen to be empty elements), then displays the results of applying the position ( ) and name ( ) functions to each of the element node children of the element node which represents the Parent element node. See Figure 13.7.

If we have a source document that is a little more complex, we can use the name ( ) function to display both element and attribute node names. Listing 13.15 is the source document.

We can then apply the stylesheet shown in Listing 13.16, which uses the name ( ) function on both elements and attributes.

Notice that the template selected by the <xsl:apply-templates> element in the main template matches any child element nodes of the element node representing the <Parent> element. We use the <xsl:for-each> element to select each attribute node for each <Daughter> or <Son> element. The context node is thus the attribute node. The <xsl:sort> element sorts the attributes on the basis of the name of the attribute nodes as

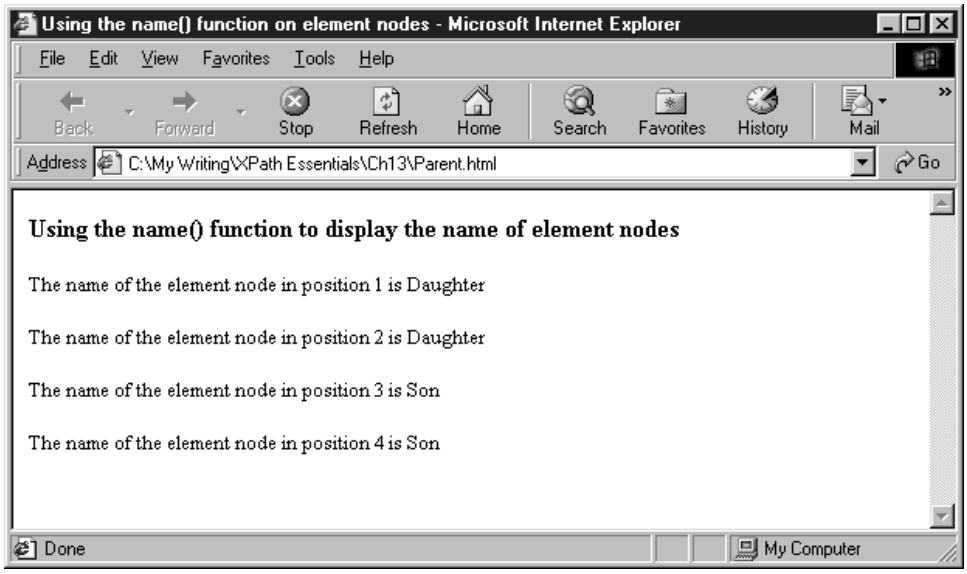

Using the name() function on element nodes - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\Parent.html
Go
Using the name() function to display the name of element nodes
The name of the element node in position 1 is Daughter
The name of the element node in position 2 is Daughter
The name of the element node in position 3 is Son
The name of the element node in position 4 is Son
Done
My Computer

Figure 13.7 Using the name ( ) Function to Display the Name of an Element Node.

```
<?xml version='1.0'?>
<Parent name="father">
<Daughter name="Martha" age="32"/>
<Daughter name="Mary" age="44"/>
<Son age="38" name="Larry"/>
<Son name="Peter" age="40"/>
</Parent>
```

Listing 13.15 A Family with Name and Age Information (ParentAttrib.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
<html>
<head>
<title>Using the name() function on element and attribute nodes</title>
</head>
<body>
<h3>Using the name() function to display the name of element and
  attribute nodes</h3>
<xsl:apply-templates select="Parent/*"/>

</body>
</html>
</xsl:template>

<xsl:template match="Daughter | Son">
<xsl:for-each select="@*">
<xsl:sort select="name(.)"/>
<p>The attribute of the <xsl:value-of select="name(..)"/> element node
  in position
<xsl:value-of select="position()"/> is <xsl:value-of select="name(.)"/>
  and has the value
<xsl:value-of select="."/></p>
</xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.16 A Stylesheet that Uses the name ( ) Function to Display Both Element and Attribute Names (ParentAttrib.xsl).

returned by the name ( ) function. Therefore, the age attribute node is processed before the name attribute node. For each of the two attributes associated with each element node, the name of its parent, `name(..)`, its position, its name, name(.), and its value are output in the HTML page, as you can see in Figure 13.8.

Using the namespace-uri( ) Function

The namespace-uri ( ) function displays the namespace URI when an element type name is a QName. The source document we will use to demonstrate this is shown in Listing 13.17.

We can use the stylesheet shown in Listing 13.18 to display the namespace URI for each of the elements contained in Listing 13.17.

The output of the transformation is shown in Figure 13.9.

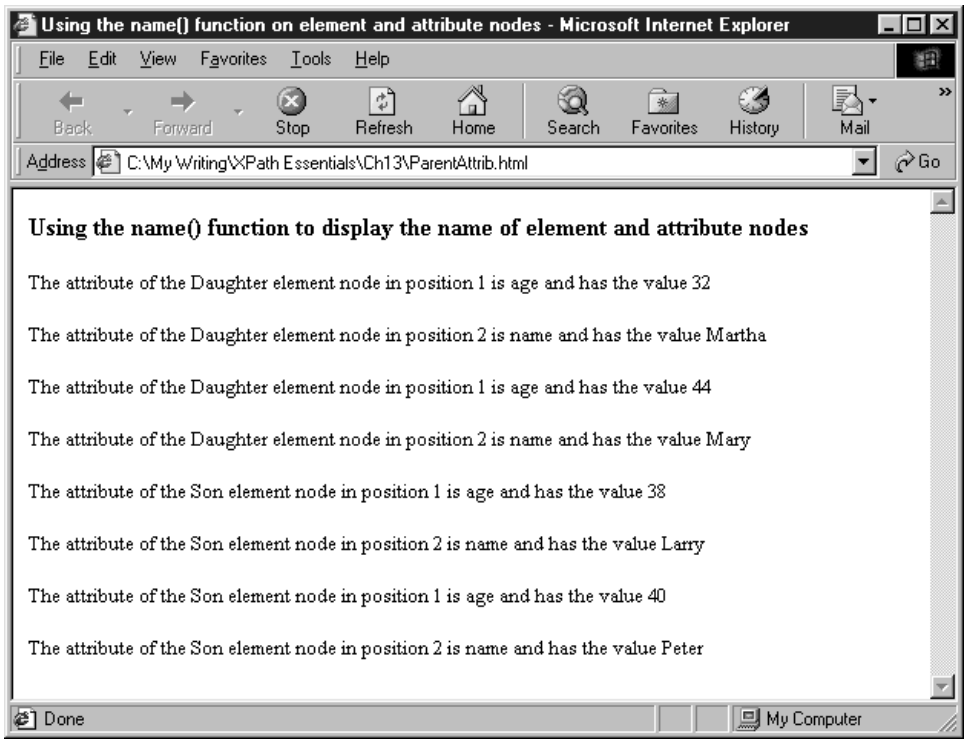

Using the name() function on element and attribute nodes - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\ParentAttrib.html
Go
Using the name() function to display the name of element and attribute nodes
The attribute of the Daughter element node in position 1 is age and has the value 32
The attribute of the Daughter element node in position 2 is name and has the value Martha
The attribute of the Daughter element node in position 1 is age and has the value 44
The attribute of the Daughter element node in position 2 is name and has the value Mary
The attribute of the Son element node in position 1 is age and has the value 38
The attribute of the Son element node in position 2 is name and has the value Larry
The attribute of the Son element node in position 1 is age and has the value 40
The attribute of the Son element node in position 2 is name and has the value Peter
Done
My Computer

Figure 13.8 Using the name ( ) Function to Sort Attributes and Then Display Element and Attribute Node Names.

```
<?xml version='1.0'?>
<Element>Some content
<XMML:Element xmlns:XMML="http://www.XMML.com/Schemas/">
Some more content.
</XMML:Element>
</Element>
```

Listing 13.17 A Simple XML Document with Namespaces (Element.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:XMML="http://www.XMML.com/Schemas/">
<xsl:output
```

Listing 13.18 A Stylesheet to Demonstrate the namespace-uri ( ) Function (Element.xsl). (*continues*)

```
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using the namespace-uri() function</title>
</head>
<body>
<table width="90%" border="2">
<tr>
<td width="40%"><b>Element Type Name</b></td>
<td width="50%"><b>Namespace URI</b></td>
</tr>
<xsl:apply-templates select="//*"/>
</table>
</body>
</html>
</xsl:template>

<xsl:template match=" Element | XMML:Element">
<tr>
<td><xsl:value-of select="name(.)"/></td>
<xsl:choose>
<xsl:when test="string-length(namespace-uri(.))>0">
<td><xsl:value-of select="namespace-uri(.)"/></td>
</xsl:when>
<xsl:otherwise>
<td> </td>
</xsl:otherwise>
</xsl:choose>
</tr>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.18 (*Continued*)

Using the position( ) Function

The position ( ) function is one of the more generally useful XPath functions. We will look at one way it can be used, with Listing 13.19 as the source document.

We can display the names of each president in the arbitrarily chosen list of U.S. presidents using Listing 13.20.

Within the <xsl:template> element, which matches on President, the code

```
<xsl:value-of select="position()"/>
```

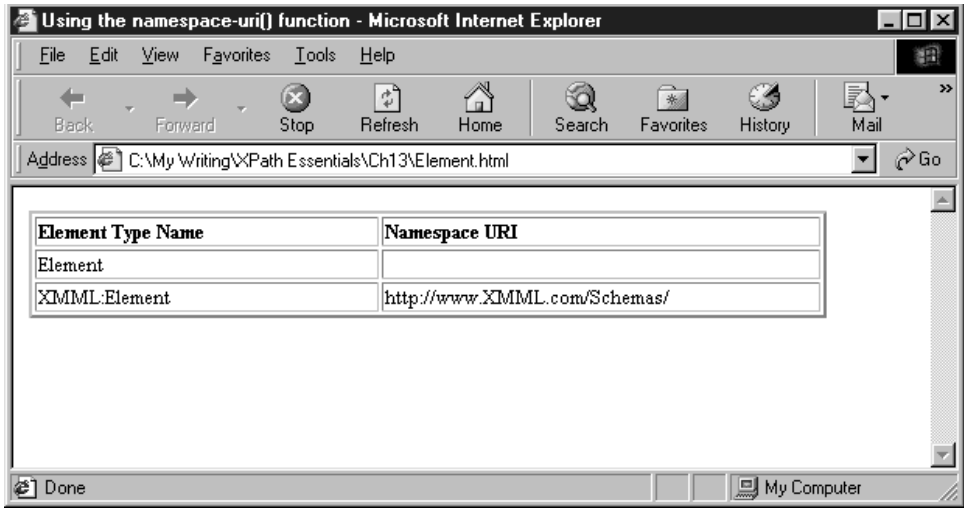


Figure 13.9 Showing the Namespace URI Using the namespace-uri ( ) Function.

```
<?xml version='1.0'?>
<Presidents>
<President>
<LastName>Lincoln</LastName>
<FirstName>Abraham</FirstName>
</President>
<President>
<LastName>Carter</LastName>
<FirstName>Jimmy</FirstName>
</President>
<President>
<LastName>Reagan</LastName>
<FirstName>Ronald</FirstName>
</President>
<President>
<LastName>Bush</LastName>
<FirstName>George</FirstName>
</President>
<President>
<LastName>Clinton</LastName>
<FirstName>William Jefferson</FirstName>
</President>
<President>
<LastName>Bush</LastName>
<FirstName>George Dubya</FirstName>
</President>
</Presidents>
```

Listing 13.19 A Brief List of U.S. Presidents (Presidents.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Displaying position in an ordered list.</title>
</head>
<body>
<h3>The presidents in the list with their position in the list:</h3>
<xsl:apply-templates select="/Presidents/President"/>
</body>
</html>
</xsl:template>

<xsl:template match="President">
<p>President <xsl:value-of select="FirstName"/><xsl:text> </xsl:text>
  <xsl:value-of select="LastName"/> is in position <xsl:value-of
  select="position()"/>.</p>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.20 A Stylesheet to Demonstrate the position ( ) Function (Presidents.xsl).

selects the position of the context node in the set of nodes selected by the <xsl:apply-templates> element of the main template.

The output of the transformation is shown in Figure 13.10.

Using Number Functions

In this section we will look briefly at some examples that use XPath number functions.

Using the sum( ) Function

The sum ( ) function allows us to add the numbers within a chosen node set. Listing 13.21 is a simple source document that allows us to examine the sum ( ) function.

Listing 13.22 makes use of the sum ( ) function to display the total cost of the line items enumerated in Listing 13.21.

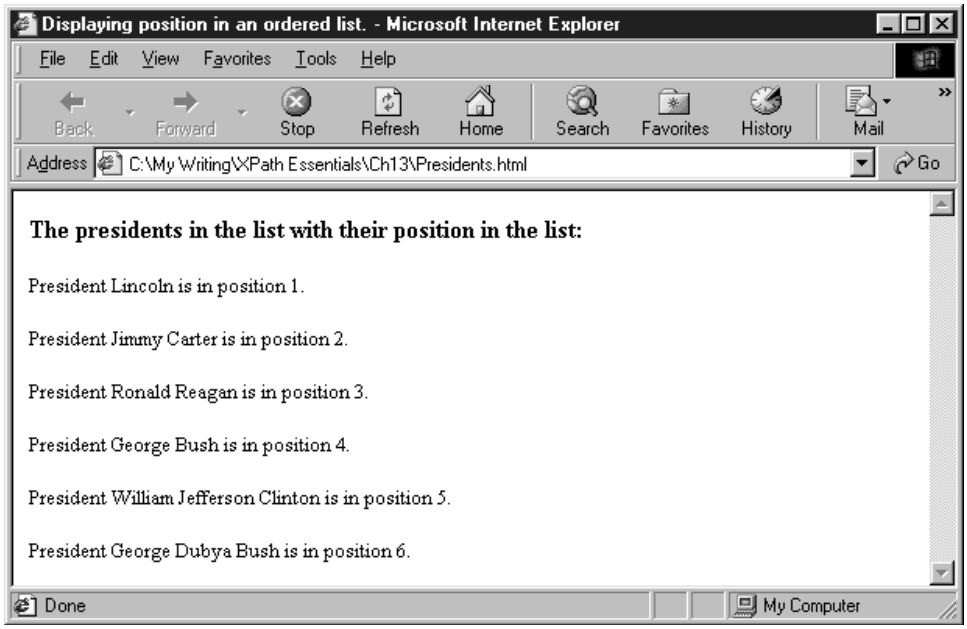

Displaying position in an ordered list. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\Presidents.html
Go
The presidents in the list with their position in the list:
President Lincoln is in position 1.
President Jimmy Carter is in position 2.
President Ronald Reagan is in position 3.
President George Bush is in position 4.
President William Jefferson Clinton is in position 5.
President George Dubya Bush is in position 6.
Done
My Computer

Figure 13.10 Displaying the Position in a List.

```
<?xml version='1.0'?>
<Purchase>
<LineItem>234</LineItem>
<LineItem>12</LineItem>
<LineItem>98</LineItem>
<LineItem>39</LineItem>
<LineItem>901</LineItem>
<LineItem>33</LineItem>
<LineItem>88</LineItem>
</Purchase>
```

Listing 13.21 A Purchase of Line Items Expressed in XML (Purchase.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
```

Listing 13.22 A Stylesheet to Demonstrate the sum ( ) Function (Purchase.xsl).

(*continues*)

```
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Finding the sum of a list of costs.</title>
</head>
<body>
<h3>Adding the cost of the line items</h3>
<p>
<xsl:apply-templates select="/Purchase/LineItem"/>
</p>
<p>
<xsl:call-template name="FindTotal"/>
</p>
</body>
</html>
</xsl:template>

<xsl:template match="LineItem">
Line Item #<xsl:value-of select="position()"/> cost $<xsl:value-of
  select="."/><br />
</xsl:template>

<xsl:template name="FindTotal">
Total cost is: $<xsl:value-of select="sum(/Purchase/LineItem)"/>.
</xsl:template>
</xsl:stylesheet>
```

Listing 13.22 (*Continued*)

The template that matches on the LineItem element nodes, instantiated using the <xsl:apply-templates> element of the main template, simply displays the cost of each item together with information about the position of the line item in the list.

The named template is where the adding up of the costs is done. The value of the <xsl:value-of> element below adds the individual values corresponding to the content of each LineItem element node. The display produced by the transformation is shown in Figure 13.11.

```
sum(/Purchase/LineItem)
```

Now let's create an example that uses both the sum ( ) and count ( ) functions. If we have a book catalog like the one in Listing 13.23, we might want to calculate the average price of the books listed in the catalog. To do that, we can add the prices of all the books and then divide by the number of books in the catalog.

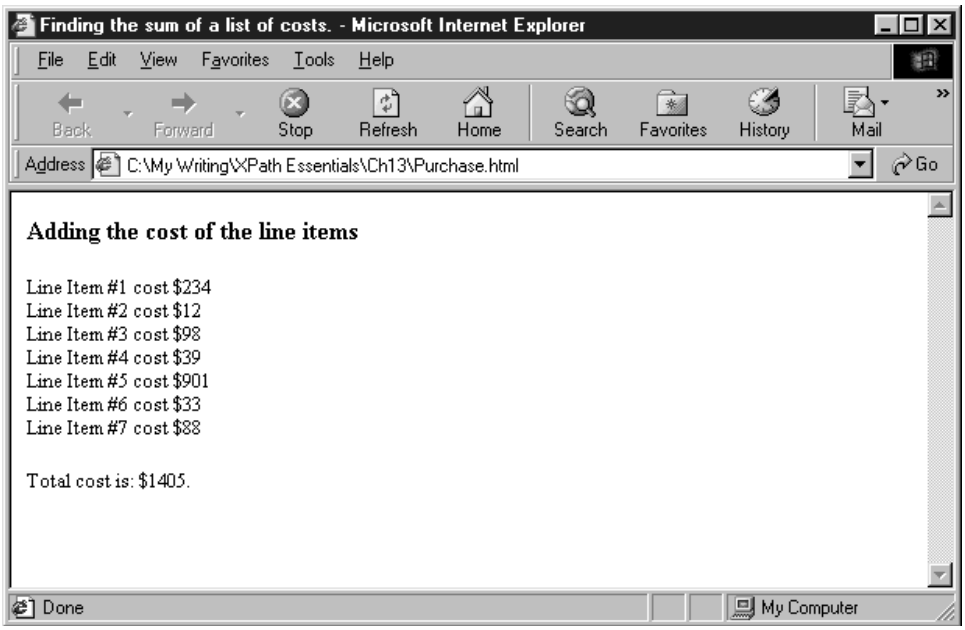

Finding the sum of a list of costs. - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\Purchase.html
Go
Adding the cost of the line items
Line Item #1 cost $234
Line Item #2 cost $12
Line Item #3 cost $98
Line Item #4 cost $39
Line Item #5 cost $901
Line Item #6 cost $33
Line Item #7 cost $88
Total cost is: $1405.
Done
My Computer

Figure 13.11 Finding the Total of a List of Costs Using the sum ( ) Function.

```
<?xml version='1.0'?>
<BookCatalog>
<Book Price="44.99">
<Title>XSL Essentials</Title>
<Authors>
  <Author>Michael Fitzgerald</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XHTML Essentials</Title>
<Authors>
  <Author>Michael Sauers</Author>
  <Author>R. Allen Wyke</Author>
</Authors>
</Book>
<Book Price="44.99">
<Title>XPath Essentials</Title>
<Authors>
  <Author>Andrew Watt</Author>
</Authors>
</Book>
```

Listing 13.23 A Catalog of Books Expressed in XML (BookCatalog2.xml).

(*continues*)

```
<Book Price="44.99">
<Title>XML Schema Essentials</Title>
<Authors>
  <Author>R. Allen Wyke</Author>
  <Author>Andrew Watt</Author>
</Authors>
</Book>
</Book>
<Book Price="49.99">
<Title>Applied XML</Title>
<Authors>
  <Author>Alex Ceponkus</Author>
  <Author>Faraz Hoodbhoy</Author>
</Authors>
</Book>
</BookCatalog>
```

Listing 13.23 (*Continued*)

The XSLT stylesheet in Listing 13.24 would list the books and then calculate an average price for them. If you look at each of the prices in the XML document, you will realize that we don't, in this case, need an XSLT stylesheet to work out the average price; however, this shows the technique.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Finding the average price of a catalog of books</title>
</head>
<body>
<h3>Wiley XML books</h3>
<table width="100%" border="2">
<tr>
<td width="40%"><b>Title</b></td>
```

Listing 13.24 A Stylesheet to Demonstrate the count ( ) and sum ( ) Functions (BookCatalog2.xsl).

```
<td width="40%"><b>Author(s)</b></td>
<td width="20%"><b>Price ($)</b></td>
</tr>
<xsl:apply-templates select="/BookCatalog/Book"/>
<xsl:call-template name="FindAverage"/>
</table>
</body>
</html>
</xsl:template>

<xsl:template match="Book">
<tr>
<td><xsl:value-of select="Title"/></td>
<xsl:choose>
<xsl:when test="count(Authors/Author) > 1">
<td><xsl:value-of select="Authors/Author[position()=1]"/> &amp;
  <xsl:value-of select="Authors/Author[position()=2]"/></td>
</xsl:when>
<xsl:otherwise>
<td><xsl:value-of select="Authors/Author"/></td>
</xsl:otherwise>
</xsl:choose>
<td><xsl:value-of select="@Price"/></td>
</tr>
</xsl:template>

<xsl:template name="FindAverage">
<tr>
<td> </td>
<td align="right"><b>Average Price:</b></td>
<td><xsl:value-of select="sum(/BookCatalog/Book/@Price)div
  count(/BookCatalog/Book)"/></td>
</tr>
</xsl:template>

</xsl:stylesheet>
```

Listing 13.24 (*Continued*)

The calculation is carried out in the named template "FindAverage". The value of the select attribute of the <xsl:value-of> element calculates the sum of the costs of the books, as reflected in the Price attribute of each <Book> element, and divides that by the count of the number of Book element nodes, therefore finding the average cost of each book. The output of the transformation is displayed in Figure 13.12.

```
sum(/BookCatalog/Book/@Price)div count(/BookCatalog/Book)
```

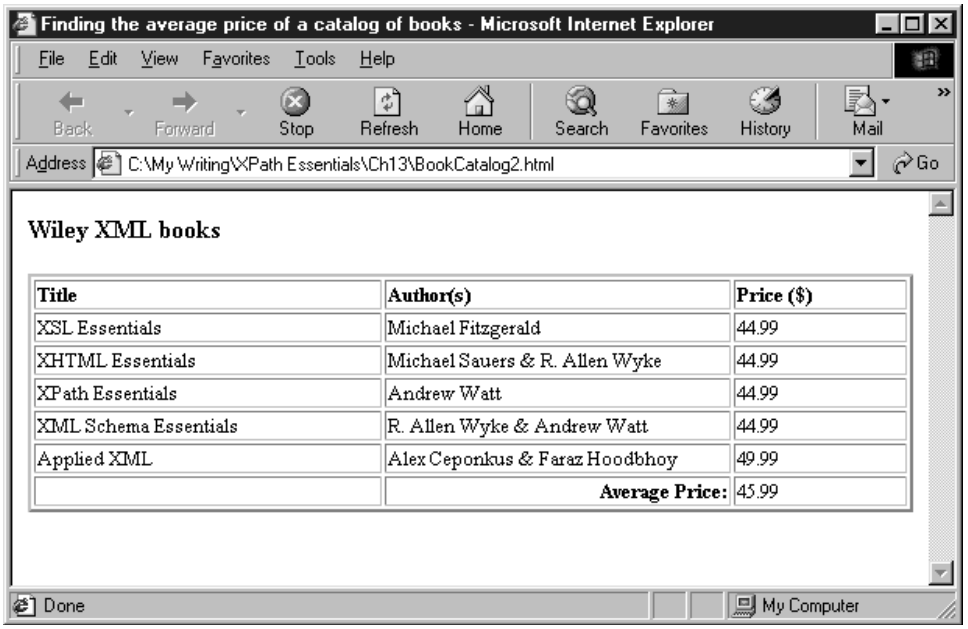


Figure 13.12 Finding the Average Cost of Books Using the sum ( ) and count ( ) Functions.

Using String Functions

In this section we will look at some examples of using string functions.

Using the concat( ) Function

The concat ( ) function combines two strings into one. Listing 13.25 shows a list of composers in which the first name and last name are held in separate elements. The concat ( ) function allows us to combine the contents of the separate elements into a single element.

```
<?xml version='1.0'?>
<Composers>
<Composer>
<LastName>Beethoven</LastName>
<FirstName>Ludwig</FirstName>
</Composer>
<Composer>
<LastName>Berlioz</LastName>
```

Listing 13.25 A List of Classical Composers in XML (Composers01.xml).

```
<FirstName>Hector</FirstName>
</Composer><Composer>
<LastName>Dvorak</LastName>
<FirstName>Antonin</FirstName>
</Composer><Composer>
<LastName>Tippett</LastName>
<FirstName>Michael</FirstName>
</Composer>
<Composer>
<LastName>Ives</LastName>
<FirstName>Charles</FirstName>
</Composer>
</Composers>
```

Listing 13.25 (*Continued*)

The stylesheet that transforms the source document and uses the concat ( ) function is shown in Listing 13.26.

The XML document output as a result of applying Listing 13.26 is shown in Listing 13.27.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<NewComposers>
<xsl:for-each select="/Composers/Composer">
<xsl:sort select="LastName"/>
<Composer><xsl:value-of select="concat(LastName, ',' , FirstName)"/>
</Composer>
</xsl:for-each>
</NewComposers>
</xsl:template>

</xsl:stylesheet>
```

Listing 13.26 A Stylesheet to Demonstrate the concat ( ) Function (Composers01.xsl).

```
<NewComposers>
   <Composer>Beethoven,Ludwig</Composer>
   <Composer>Berlioz,Hector</Composer>
   <Composer>Dvorak,Antonin</Composer>

   <Composer>Tippett,Michael</Composer>
   <Composer>Ives,Charles</Composer>
</NewComposers>
```

Listing 13.27 An XML List of Composers Following Use of the concat ( ) Function (Composers01Out.xml).

Using the contains( ) Function

If you want to select an element or attribute on the basis of some part of its content, you can use the contains ( ) function.

We can use Listing 13.28 to demonstrate the use of the contains ( ) function.

The XSLT stylesheet in Listing 13.29 demonstrates how to use the contains ( ) function to select an element that contains a particular name.

```
<?xml version='1.0'?>
<Travel>
<Flight>
<Person>Peter Gray</Person>
<From>New York</From>
<To>London</To>
<FlightNo>ABC123</FlightNo>
<Class>Business</Class>
<Date>2002-08-11</Date>
</Flight>
<Flight>
<Person>Karen Smith</Person>
<From>Tokyo</From>
<To>Paris</To>
<FlightNo>TKY199</FlightNo>
<Class>Business</Class>
<Date>2003-04-09</Date>
</Flight>
<Flight>
<Person>Carol Peters</Person>
<From>San Francisco</From>
<To>New York</To>
<FlightNo>DBE890</FlightNo>
<Class>Economy</Class>
<Date>2002-02-04</Date>
```

Listing 13.28 An XML List of Journeys by Person (Travel.xml).

```
</Flight>
<Flight>
<Person>Peter Gray</Person>
<From>London</From>
<To>Paris</To>
<FlightNo>BAC876</FlightNo>
<Class>Economy</Class>
<Date>2001-12-23</Date>
</Flight>
<Flight>
<Person>Chloe Jamna</Person>
<From>Rome</From>
<To>Toronto</To>
<FlightNo>ACD789</FlightNo>
<Class>Business</Class>
<Date>2002-08-31</Date>
</Flight>
</Travel>
```

Listing 13.28 (*Continued*)

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Selecting elements by content</title>
</head>
<body>
<h3>An example of selecting elements by content - </h3>
<xsl:apply-templates select="Travel/Flight/Person[.=contains(.,
  'Karen')]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Person">
<h3>Flight for <xsl:value-of select="."/>:</h3>
<p><b>Trip from: </b> <xsl:value-of select="../From"/></p>
```

Listing 13.29 A Stylesheet to Demonstrate the contains ( ) Function (TravelContains.xsl). (*continues*)

```
<p> <b>Trip to: </b> <xsl:value-of select="../To"/></p>
<p><b>Class: </b> <xsl:value-of select="../Class"/></p>
<p><b>Date: </b> <xsl:value-of select="../Date"/></p>
<br />
</xsl:template>
</xsl:stylesheet>
```

Listing 13.29 (*Continued*)

The select attribute of the <xsl:apply-templates> element in the main template

```
<xsl:apply-templates select="Travel/Flight/Person[.=contains(.,
'Karen')]"/>
```

includes a predicate `[.=contains(., 'Karen')]`, which means that only if the value contained in the self node (remember that the full stop is the abbreviated syntax for `self::node()`) is a Person element node then it is the selected node for processing. Thus the <xsl:template> element that matches on a Person element node is instantiated only if the value of that element node contains the word "Karen".

We can also use the contains ( ) function in a slightly different way to achieve much the same thing. Listing 13.30 shows a number of Wiley XML books from which we will select and display titles from the XML Essentials series.

```
<?xml version='1.0'?>
<BookCatalog>
<Book>
<Title>XSL Essentials</Title>
<Authors>
<Author>Michael Fitzgerald</Author>
</Authors>
</Book>
<Book>
<Title>XHTML Essentials</Title>
<Authors>
<Author>Michael Sauers</Author>
<Author>R. Allen Wyke</Author>
</Authors>
</Book>
<Book>
<Title>XPath Essentials</Title>
<Authors>
<Author>Andrew Watt</Author>
</Authors>
</Book>
<Book>
<Title>XML Specification Guide</Title>
```

Listing 13.30 An Abbreviated List of Wiley XML Books (BookCatalog.xml).

```
<Authors>
<Author>Ian S. Graham</Author>
<Author>Liam Quin</Author>
</Authors>
</Book>
<Book>
<Title>XML Schema Essentials</Title>
<Authors>
<Author>R. Allen Wyke</Author>
<Author>Andrew Watt</Author>
</Authors>
</Book>
<Book>
<Title>Applied XML</Title>
<Authors>
<Author>Alex Ceponkus</Author>
<Author>Faraz Hoodbhoy</Author>
</Authors>
</Book>
</BookCatalog>
```

Listing 13.30 (*Continued*)

We can use the XSLT stylesheet in Listing 13.31 to test whether or not the title of a book contains the word "Essentials". If it does, then we assume that the book belongs in the XML Essentials series and information about it should be displayed.

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>XML Essentials Series books</title>
</head>
<body>
<h3>Books in the catalog which are from the XML Essentials series.</h3>
<p>
<xsl:apply-templates select="/BookCatalog/Book[contains(Title,
  'Essentials')]"/>
```

Listing 13.31 A Stylesheet to Demonstrate the contains ( ) Function (BookCatalog.xsl). (*continues*)

```
</p>
</body>
</html>
</xsl:template>

<xsl:template match="Book">
<b><xsl:value-of select="Title"/></b> by
<xsl:choose>
<xsl:when test="count(Authors/Author) &gt; 1">
<xsl:if test="count(Authors/Author) = 2">
<xsl:value-of select="Authors/Author[1]"/>
<xsl:text> &amp; </xsl:text>
<xsl:value-of select="Authors/Author[2]"/><br />
</xsl:if>
</xsl:when>
<xsl:otherwise>
<xsl:value-of select="Authors/Author"/><br />
</xsl:otherwise>

</xsl:choose>
<br/>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.31 (*Continued*)

It is the select attribute of the <xsl:apply-templates> element in the main template

```
<xsl:apply-templates select="/BookCatalog/Book[contains(Title,
  'Essentials')]"/>
```

that determines that only books with the word "Essentials" in the title are selected for further processing.

Note that the <xsl:template> element, which matches on Book element nodes, contains an <xsl:choose> to vary the output, depending on the number of authors of a book.

The output from the transformation, shown in Figure 13.13, demonstrates that books with a title containing the word "Essentials" are selected for display.

Using the starts-with( ) Function

Sometimes it is useful to select an element or some other part of an XML source document if the element type name, or some other name or string, starts with a particular sequence of characters.

Listing 13.32 contains a list of some famous composers and we will use the starts-with function to select a number of those only when their surname starts with the letters "Be".

Listing 13.33 contains a stylesheet that uses the starts-with ( ) function to select composers for processing in the <xsl:template> element that matches on Composer element nodes.

The output of the transformation is shown in Figure 13.14.

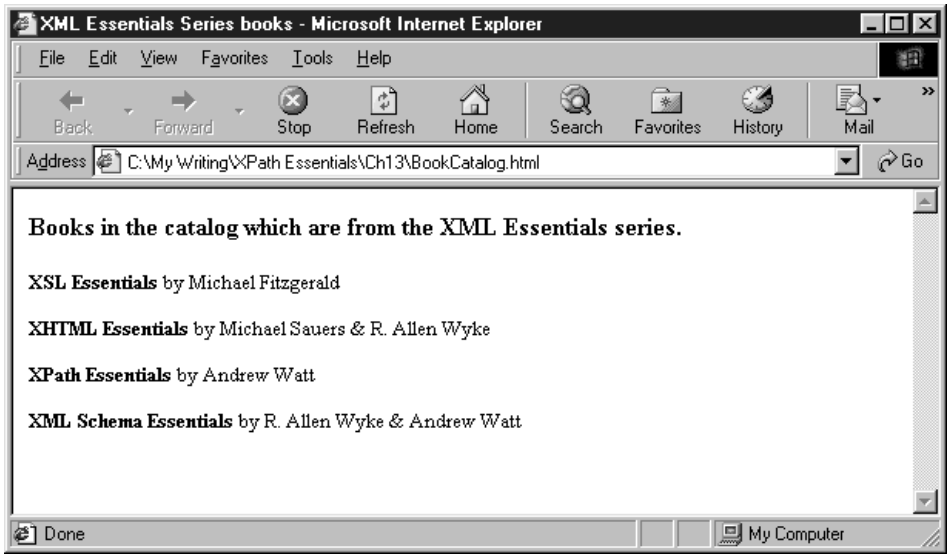

XML Essentials Series books - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\BookCatalog.html
Go
Books in the catalog which are from the XML Essentials series.
XSL Essentials by Michael Fitzgerald
XHTML Essentials by Michael Sauers & R. Allen Wyke
XPath Essentials by Andrew Watt
XML Schema Essentials by R. Allen Wyke & Andrew Watt
Done
My Computer

Figure 13.13 Selecting Books for Display If Their Title Contains the Word "Essentials".

```
<?xml version='1.0'?>
<Composers>
<Composer>
<LastName>Beethoven</LastName>
<FirstName>Ludwig</FirstName>
<Nationality>German</Nationality>
</Composer>
<Composer>
<LastName>Berlioz</LastName>
<FirstName>Hector</FirstName>
<Nationality>French</Nationality>
</Composer><Composer>
<LastName>Dvorak</LastName>
<FirstName>Antonin</FirstName>
<Nationality>Czech</Nationality>
</Composer><Composer>
<LastName>Tippett</LastName>
<FirstName>Michael</FirstName>
<Nationality>British</Nationality>
</Composer>
<Composer>
<LastName>Ives</LastName>
<FirstName>Charles</FirstName>
<Nationality>American</Nationality>
</Composer>
</Composers>
```

Listing 13.32 A List of Classical Composers Expressed in XML (Composers02.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using the starts-with() function</title>
</head>
<body>
<h3>Composers whose names begin with "Be"</h3>
<xsl:apply-templates select="/Composers/Composer[starts-with(.,
'Be')]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Composer">
<p><xsl:value-of select="LastName"/>, <xsl:value-of select="FirstName"/>
  was <xsl:value-of select="Nationality"/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.33 A Stylesheet to Demonstrate the starts-with ( ) Function (Composers02.xsl).

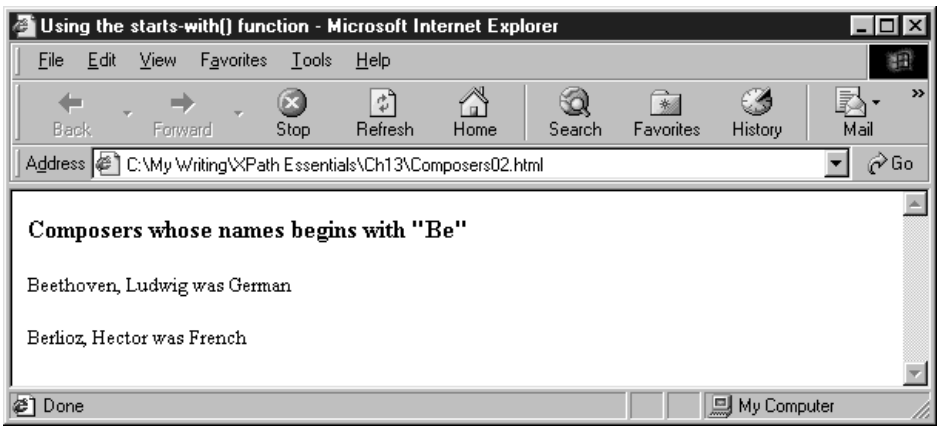

Using the starts-with() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\Composers02.html
Go
Composers whose names begins with "Be"
Beethoven, Ludwig was German
Berlioz, Hector was French
Done
My Computer

Figure 13.14 Using the starts-with ( ) Function.

Using Boolean Functions

In this section we will look briefly at some examples using Boolean functions.

Using the boolean( ) Function

The boolean ( ) function converts its argument to a Boolean value. Let's look at the boolean ( ) function in action using Listing 13.34 as our XML source document.

The XSLT stylesheet in Listing 13.35 will use the boolean ( ) function to convert the content of the various <Thing> elements to a Boolean value.

```
<?xml version='1.0'?>
<ToBoolean>
<Thing something="A string">Marty</Thing>
<Thing something="A number">3</Thing>
<Thing something="The number zero">0</Thing>
<Thing something="An empty string"></Thing>
<Thing something="A boolean">true</Thing>
</ToBoolean>
```

Listing 13.34 A Number of XML Elements Whose Content Will Be Converted Using the boolean ( ) Function (ToBoolean.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using the boolean() function</title>
</head>
<body>
<h3>The content of each &lt;Thing&gt; element is changed to a boolean
  value.</h3>
<xsl:apply-templates select="//Thing"/>
</body>
</html>
```

Listing 13.35 A Stylesheet to Demonstrate the boolean ( ) Function (ToBoolean.xsl).

(*continues*)

```
</xsl:template>

<xsl:template match="Thing">
<xsl:choose>
<xsl:when test="contains(@something, 'number')">
<p>When <xsl:value-of select="."/> which is <xsl:value-of
  select="@something"/> is converted to
a boolean value the value it takes is: <xsl:value-of
  select="boolean(number(.))"/>.</p>
</xsl:when>
<xsl:when test="contains(@something, 'string')">
<p>When <xsl:value-of select="."/> which is <xsl:value-of
  select="@something"/> is converted to
a boolean value the value it takes is: <xsl:value-of
  select="boolean(string(.))"/>.</p>
</xsl:when>
<xsl:otherwise>
<p>When <xsl:value-of select="."/> which is <xsl:value-of
  select="@something"/> is converted to
a boolean value the value it takes is: <xsl:value-of
  select="boolean(.)"/>.</p>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.35 (*Continued*)

Notice that the business end of the stylesheet has an <xsl:choose> element nested within the template. I did that so we could convert the content of each <Thing> element explicitly to a particular type of value.

Those <Thing> elements that had the word "number" in the something attribute were converted explicitly to numbers using the number ( ) function within an <xsl:when> element:

```
<xsl:when test="contains(@something, 'number')">
<p>When <xsl:value-of select="."/> which is <xsl:value-of
select="@something"/> is converted to
a boolean value the value it takes is: <xsl:value-of
select="boolean(number(.))"/>.</p>
</xsl:when>
```

Similarly, when the word "string" appeared within the something attribute of a <Thing> element, the string() function was used to convert the content of the element to a string:

```
<xsl:when test="contains(@something, 'string')">
<p>When <xsl:value-of select="."/> which is <xsl:value-of
  select="@something"/> is converted to
```

```
a boolean value the value it takes is: <xsl:value-of
  select="boolean(string(.))"/>.</p>
</xsl:when>
```

When the content was already a Boolean value, I used the <xsl:otherwise> to process the node set containing the context node without using either the number ( ) or string ( ) functions:

```
<xsl:otherwise>
<p>When <xsl:value-of select="."/> which is <xsl:value-of
  select="@something"/> is converted to
a boolean value the value it takes is: <xsl:value-of
  select="boolean(.)"/>.</p>
</xsl:otherwise>
```

Using the not( ) Function

The not ( ) function allows us to negate a value or it allows us to select on the opposite of a stated condition. Listing 13.36 shows a simple source document, which will allow us to look at this function.

Let's decide to display paragraphs that are not narratives. Listing 13.37 shows how we can use the not ( ) function to do that.

```
<?xml version='1.0'?>
<Paragraphs>
<Paragraph type="Caution">Beware!</Paragraph>
<Paragraph type="Narrative">Here is a story.</Paragraph>
<Paragraph type="Caution">Be very careful!</Paragraph>
<Paragraph type="Narrative">Mary had a little lamb.</Paragraph>
</Paragraphs>
```

Listing 13.36 A Brief Collection of Simple Paragraphs Expressed in XML (Paragraphs.xml).

```
<?xml version='1.0'?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output
  method="html"
  indent="yes"
  />
<xsl:strip-space elements="*"/>

<xsl:template match="/">
<html>
<head>
<title>Using the not() function</title>
```

Listing 13.37 A Stylesheet to Demonstrate the not ( ) Function (Paragraphs.xsl).

(*continues*)

```
</head>
<body>
<h3>Selecting paragraphs to display which are NOT narrative
  paragraphs.</h3>
<xsl:apply-templates
select="/Paragraphs/Paragraph[not(@type='Narrative')]"/>
</body>
</html>
</xsl:template>

<xsl:template match="Paragraph">
<p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

Listing 13.37 (*Continued*)

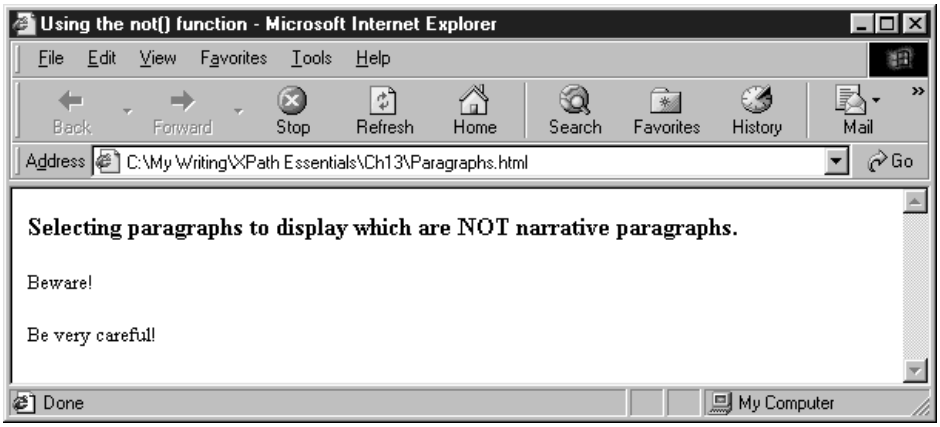

Using the not() function - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\My Writing\XPath Essentials\Ch13\Paragraphs.html
Go
Selecting paragraphs to display which are NOT narrative paragraphs.
Beware!
Be very careful!
Done
My Computer

Figure 13.15 Using the not ( ) Function.

The select attribute of the <xsl:apply-templates> element in the main template has a predicate, [not(@type='Narrative')], which means that only Paragraph element nodes that do not have a type attribute with a value of "Narrative" are selected for display.

The output of the transformation is shown in Figure 13.15.

Looking Ahead

In each of the chapters up to this point we have looked at XPath 1.0 and its characteristics. In Chapter 14 we will look ahead to XPath version 2.0, which is currently under development, in particular the foundational work that is being carried out at the World Wide Web Consortium to use XPath with the XML Query Language (XQuery).

XPath 2.0 and XQuery

One of the most important emerging areas for the future use of XPath is as a foundation for, and in conjunction with, the XML Query Language (XQuery) currently being developed at the W3C. The future development of the XPath 2.0 specification and the development of the XQuery 1.0 specification, currently at Working Draft status, are going to be intimately intertwined.

In addition to the relationship with XQuery 1.0, XPath 2.0 will also be closely intertwined with the XSLT 2.0 specification.

NOTE **At the time of writing, the details of the future of XQuery are emerging rapidly from the W3C with successive multiple Working Drafts. Be sure to check the material in this chapter against the version of the XPath 2.0 and XQuery 1.0 specifications current at the time you are reading this book. URLs are given at the end of the chapter.**

XPath 2.0 Working Drafts

At the time of writing, the development of the XPath 2.0 specification is at an early stage and is very fluid, in part because of the need to accommodate XPath 2.0 to the requirements of XQuery.

Currently, two W3C Working Drafts have begun the process of defining exactly what XPath 2.0 will contain. The two drafts are

- XPath 2.0 Requirements (www.w3.org/TR/2001/WD-xpath20req-20010214)
- XQuery 1.0 and XPath 2.0 Data Model (www.w3.org/TR/2001/WD-query-datamodel-20010607/)

No Working Draft of the specification proper for XPath 2.0 has yet been released by the W3C—only a Working Draft of the Requirements document.

XPath 2.0 Requirements

As you have seen earlier in this book, XPath 1.0 provides much useful functionality for selecting XML elements in source documents for use with XSLT and, potentially, with XPointer. However, since the XML 1.0 Recommendation was first issued in February 1998 and the XPath Recommendation emerged in November 1999, the range of uses for XML has expanded rapidly. To meet the needs of these newer uses of XML, the W3C has been developing many other XML-related specifications. As a result, while XPath 1.0 was adequate to meet many needs in late 1999, it is beginning to show its limitations.

NOTE The XPath 2.0 Requirements Working Draft is located at www.w3.org/TR/xpath20req.

In the XPath 2.0 Working Draft (current at the time of writing), the following explicit goals for XPath 2.0 were stated:

- Simplify manipulation of XML-Schema typed content
- Simplify manipulation of string content
- Support related XML standards
- Improve ease of use
- Improve interoperability
- Improve internationalization support
- Maintain backward compatibility
- Enable improved processor efficiency

Let's look at each of these in turn.

Manipulation of XML-Schema Typed Content

At the time that the XPath 1.0 Recommendation was produced, there was no Recommendation for XML Schema. The only type of schema for XML was the Document Type Definition described in the XML 1.0 Recommendation; although, of course, a DTD was not written in XML syntax. Therefore, XPath 1.0 has no awareness of XSD (XML) Schema, its datatypes, or its constraints on XML documents.

NOTE **The XML Schema specification became a Recommendation in May 2001 published by the W3C in three parts. Part 0 is a primer and is located at www.w3.org/TR/2001/REC-xmlschema-0-20010502/. Part 1 describes XML Schema structures and is located at www.w3.org/TR/2001/REC-xmlschema-1-20010502/. Part 2 describes datatypes and is located at www.w3.org/TR/2001/REC-xmlschema-2-20010502/.**

XML Schema, sometimes called XSD Schema to distinguish it from other schemas written in XML, is a pivotal part of the W3C's strategy for the future of XML. Thus it is seen as essential that XPath in version 2.0 adds Schema-aware functionality. The book XML Schema Essentials in this series provides an introduction to W3C XML Schema.

XPath 1.0 has four data types: node set, string, Boolean, and number. The XML Schema specification lists 19 datatypes:

- *string*—character strings in XML
- *boolean*—binary value logic
- *decimal*—arbitrary precision decimal numbers
- *float*—IEEE single-precision floating point number
- *double*—IEEE double-precision floating point number
- *duration*—a duration of time
- *dateTime*—a specific instant of time
- *time*—an instant of time that recurs every day
- *date*—a calendar date
- *gYearMonth*—a specific Gregorian month in a specific Gregorian year
- *gYear*—a Gregorian calendar year
- *gMonthDay*—a Gregorian date, such as June 6, which recurs
- *gDay*—a Gregorian day which recurs, such as the sixth of the month
- *gMonth*—a Gregorian month, which recurs every year
- *hexBinary*-arbitrary hex-encoded binary data
- *base64Binary*—Base64-encoded arbitrary binary data
- *anyURI*—a Uniform Resource Identifier
- *QName*—a qualified name
- *NOTATION*—a qualified name

The addition of support for XML Schema datatypes will be a significant addition to the capabilities of XPath 2.0.

XML Schema allows decimals to have a leading plus sign. Thus it is intended that XPath 2.0 will support a unary plus operator to accommodate that aspect of XML Schema.

XPath 2.0 will be required to provide casting and constructor functions to enable users to cast and construct XML Schema primitive datatypes.

XML Schema: Structures provide the ability to define an element or attribute as being of a particular type. XPath 2.0 is expected to be able to test the type of an element or attribute.

XML Schema provides the ability to define a hierarchy of types by derivation. XPath 2.0 is expected to be able to select instances of a type, including types derived by restriction or extension from that type.

XML Schema: Datatypes provide the ability to include two or more element names in a substitution group. XPath 2.0 is expected to be able to test whether or not an element is a member of an XML Schema substitution group.

Manipulation of String Content

User experience in using XPath 1.0 with XSLT 1.0 has led to requests for additional functionality to manipulate strings. It is likely that string functions, which allow padding of strings, conversion of case within strings, and string replacement will be added to XPath 2.0.

Additional functionality will be required for the use of regular expressions.

Support Related XML Standards

One of the most important related standards to be supported by XPath 2.0 is XQuery 1.0. In addition XPath 2.0 will be required to provide a basis for XSLT 2.0. The scope of XPath 2.0 functionality must incorporate the expression language for XQuery 1.0, XSLT 2.0, and also, likely, XPointer.

Similarly, as just mentioned, XPath 2.0 will be required to accommodate the arrival of XML Schema.

The typed value of an element can be null. XPath 2.0 will be required to define the behavior of operations applied to null values.

The XML Infoset is an abstract data model likely to be very influential in the future for many XML-related specifications. XPath 2.0 is required to be expressed in terms of the XML Information Set (see the XPath 2.0 Data Model section later in this chapter).

Improve Ease of Use

The XPath 2.0 Requirements document lists several ways in which XPath 2.0 is expected to be more easily usable than XPath 1.0.

It must be possible for Boolean expressions involving node sets to be able to express the notions of “for all” and “for any.”

XPath 1.0 supports the sum ( ) and count ( ) functions. XPath 2.0 will be required to add min ( ) and max ( ) functions to return, respectively, the minimum and maximum values in a node set.

XPath 1.0 provides the means to achieve the union of two node sets. XPath 2.0 is intended to add functionality to express the intersection and difference of two node sets.

In XPath 1.0 neither unions nor node-set functions are permitted to appear after a “/” in an XPath location path. To provide improved alignment with XPointer and to improve ease of writing location paths that offer multiple alternatives, it is anticipated that the

current restrictions in XPath 1.0 will be loosened. Allowing node-set functions to appear after a "/" would permit better access into external documents when using the XSLT document ( ) function.

XPath 2.0 must provide a conditional expression that takes three expressions, so that if the first expression evaluates to "true," then the second expression is evaluated; but if the first expression evaluates to "false," then the third expression is evaluated.

Improve Interoperability

XPath 2.0 will be required to support the operators and type-coercion rules currently under development by a joint XSLT/Schema/Query task force at the W3C.

XPath 2.0 is expected to support a list datatype, specifically an ordered list of simple-type values.

Improve Internationalization Support

The XPath 2.0 Requirements Working Draft does not yet specify how XPath 2.0 is to improve support for internationalization.

Maintain Backward Compatibility

It is the intention that any XPath 1.0 expression will also be a valid XPath 2.0 expression, but the Working Draft recognizes that this may not be possible in every case. Therefore, the intention is to maximize backward compatibility, but there may be situations where an XPath 1.0 expression or location path is no longer valid in XPath 2.0.

Enable Improved Processor Efficiency

How this is to be achieved is not yet specified.

XPath 2.0 Data Model

As mentioned in Chapter 3, the W3C has produced three data models, which are not, in their current versions, mapped transparently from one to the other. The XML Information Set data model looks like it will be the dominant one in W3C's future plans, and therefore XPath 2.0 will be required to make adaptations to accommodate compliance with the XML Information Set specification as well as with the needs of the XQuery 1.0 specification. A further limitation of each of the three data models at the time of writing is that none is XML Schema-aware.

With the release in June 2001 of a joint Working Draft for the XQuery 1.0 and XPath 2.0 data model, the intimate connection between the future of XPath and the emerging XQuery standard becomes clearer.

NOTE The Working Draft for the XQuery 1.0 and XPath 2.0 Data Model is located at www.w3.org/TR/query-datamodel/.

The Working Draft for the XPath 2.0 data model indicates that a number of significant changes are almost certain to be made to the XPath 1.0 data model that was presented in Chapter 3.

One proposed change is that the number of node types in XPath 2.0 will increase from seven in XPath 1.0 to eight. The eight proposed node types in XPath 2.0 are

- Document
- Element
- Attribute
- Text
- Namespace
- Processing instruction
- Comment
- Reference

There are two new node types. The document node fulfils functions similar to, but not identical to, the root node of XPath 1.0. A tree whose root node is a document node is referred to as a *document*. If the root node of a tree is anything other than a document node, then the tree is referred to as a *fragment*.

NOTE Although in the current draft (June 2001), a root node is not a recognized type, the specification continues to make reference to the concept and use the term *root node*. Presumably a later draft will remedy that seeming inconsistency.

The reference node is provided as a general mechanism for referring to arbitrary nodes and preserving their identity.

In addition to the likely introduction of two new node types, the XPath 2.0 data model is also likely to add

- Support for XML Schema types
- Representation of collections of documents and of Simple Values and Complex Values
- Representation of references

Values in the data model fall into five categories:

- *Nodes*—the eight kinds mentioned earlier
- *Simple values*—the union of all the value spaces of XML Schema Simple Types
- *Sequences*—an ordered collection of nodes, of simple values, or of any combination of nodes and simple values
- *Error*—the *error* value
- *Schema components*—the type of element nodes, attribute nodes, and simple values

In XPath 1.0, namespace nodes have parents. In XPath 2.0, namespace nodes do not have parents.

Data Typing in XPath 2.0

The XPath 2.0 data model specifies that element and attribute nodes contain a typed value.

Accessors

XPath 2.0 indicates that accessors are defined for all eight types of nodes mentioned earlier. Several accessors are specified:

- Node-kind
- Name
- Parent
- String-value
- Info-item kind

There is a node-kind accessor that returns a string value representing the node's kind. The possible values for the string returned are "document," "element," "attribute," "text," "namespace," "processing-instruction," "comment," or "reference."

The name accessor returns the empty sequence (if a node has no name) or it returns a sequence that contains one expanded QName. The expanded QName is in the value space of xsd:QName, as defined in the XML Schema Recommendations.

The parent accessor returns the empty sequence if the node has no parent (as in the case of the document node and namespace nodes) or otherwise returns a sequence containing one node, being the parent node of the context node.

The string-value accessor returns the string-value of the node. The string-value of the node may be part of the node or it may be computed from the string-values of the nodes' descendant nodes.

The info-item kind accessor returns a string value returning an information item's kind. The term "information item" is taken from the XML Information Set specification, which, at the time of writing, is at Candidate Recommendation status, located at www.w3.org/TR/2001/CR-xml-infoset-20010514.

The XML Information set defines a data model in terms of an information set, often termed the infoset, which consists of 11 information items:

- Document item
- Element item
- Attribute item
- Processing instruction item
- Unexpanded entity item
- Character item

- Comment item
- Doctype item
- Unparsed entity item
- Notation item
- Namespace item

NOTE Any update to the XML Infoset Candidate Recommendation of May 2001 will be located at www.w3.org/TR/xml-infoset.

The Need for XQuery

As the volume of data held as XML has increased markedly, the need to be able to query that information has become more important.

XML can be used to store a diversity of data—in relational databases, in structured or semi-structured documents, or in object repositories. The requirements for a query language for use on each type of data may vary, but XQuery is designed to be suitable for querying on all types of XML data.

Keeping Up-to-Date

As indicated earlier, the subject matter for this chapter is particularly fluid. The following URLs are where future drafts of the XPath 2.0 and XQuery 1.0 specification are likely to be located.

NOTE The W3C has changed the way in which combinations of drafts of XQuery and XPath have been grouped. Therefore, if the URLs given in this section do not give a live link, alternative places to look for information are the Technical Reports Page on the W3C Web site, located at www.w3.org/TR/. Failing that, go to the W3C home page, located at www.w3.org, and look for the links to the topics of XPath and XQuery, currently located at the bottom left of the home page.

Future versions of XPath-related drafts are likely to be located at:

- XPath 2.0 Requirements (www.w3.org/TR/xpath20req).
- XQuery and XPath 2.0 Data Model (www.w3.org/TR/query-datamodel/).
- XPath 2.0 (www.w3.org/TR/xpath20). Currently there is no page at that URL, but that is likely where the first Working Draft of the XPath 2.0 specification will be placed.

Future versions of the drafts related to the XQuery specification will likely be located at:

- XML Query Requirements (www.w3.org/TR/xmlquery-req)
- XQuery 1.0 : An XML Query Language (www.w3.org/TR/xquery)
- XML Query Use Cases (www.w3.org/TR/xmlquery-use-cases)
- XQuery 1.0 Formal Semantics (www.w3.org/TR/query-semantics/)
- XQuery 1.0 and XPath 2.0 Data Model (www.w3.org/TR/query-datamodel/)
- XML Syntax for XQuery 1.0 (XQueryX) (www.w3.org/TR/xqueryx)

I expect significant changes in the details during the period of development of XPath 2.0 and XQuery 1.0. If detailed knowledge of progress is important to you, the above URLs provide several entry points into this important evolving area.

XSLT 2.0

XPath 2.0 will form a basis for version 2.0 of XSLT, just as it forms a basis for version 1.0 of the XML Query Language.

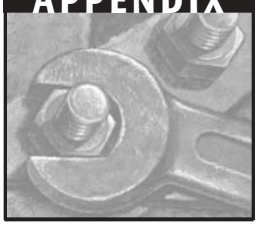
The development of XSLT 2.0 is at an early stage at W3C. The Requirements document for XSLT 2.0 has been published and is located at www.w3.org/TR/2001/WD-xslt20req-20010214. Any updates to that document will be located at www.w3.org/TR/xslt20req.

NOTE **At the time of writing, no Working Draft has been published by W3C for XSLT 2.0 itself. If W3C follows its usual naming pattern for specifications under development, you can expect to find any Working Drafts or later versions of the XSLT 2.0 specification located at www.w3.org/TR/xslt20.**

Conclusion

The future for XPath looks bright. As you have seen in this book, the range of XML technologies where it is already used and where it is likely to be used is growing. The skills and knowledge you have built as you have worked through this book should continue to be important in your XML skill-set for a considerable time to come.

APPENDIX



Online Resources

This Appendix will point you to some useful online resources on XPath and associated technologies.

World Wide Web Consortium

The main Web page for the W3C is located at www.w3.org. From links near the top of the home page you can link to a page that lists all W3C Technical Reports, www.w3.org/TR/. Additionally, there is a link to all current W3C Activity statements, so you can grasp an overview—or the details, if you prefer—of W3C's current specification development activities. This is located at www.w3.org/Consortium/Activities.

The Recommendation of November 16, 1999, for XPath 1.0 is located at www.w3.org/TR/xpath.

The Recommendation for the November 16, 1999, Recommendation for XSLT 1.0 is located at www.w3.org/TR/xslt.

It is unlikely that Recommendations for XPath 2.0 or XSLT 2.0 will be finalized for some considerable time after the publication of this book. However, the Requirements Working Drafts for XPath 2.0 and XSLT 2.0 have been published by W3C and are located, respectively, at www.w3.org/TR/xpath20req and www.w3.org/TR/xslt20req. At the time of writing there is no Working Draft of either specification (as opposed to the Requirements Working Drafts just mentioned), although a Working Draft of the XPath 2.0 Data

Model, entitled "XQuery 1.0 and XPath 2.0 Data Model" is located at www.w3.org/TR/query-datamodel/.

XPath Sites

The W3C site lacks, for XPath, the kind of overview page that is provided for some other technologies. Perhaps an overview will be provided for XPath as development of XPath 2.0 gets seriously under way. Check out the XPath link from the W3C home page, www.w3.org, to see if one has been provided. Currently the only link is to the November 1999 specification.

The vbxml.com Web site has an online XPath Reference located at www.vbxml.com/xsl/XPathRef.asp.

W3C has a mailing list for comments on the XPath specification. This has been relatively inactive for some time, but following the publication of the XPath 2.0, the Requirements Working Draft is likely to become more active during the lifetime of this book as the complex issues involved in defining the detail of the XPath 2.0 specification are discussed. To subscribe to the comments mailing list, send an email to www.xpath-comments-request@w3.org with SUBSCRIBE in the subject line of the email. Archives are available at http://lists.w3.org/Archives/Public/www-xpath-comments/.

XSLT Sites

The XSLT 1.0 Recommendation is located at www.w3.org/TR/xslt.

The XSLT.com Web site provides a useful assortment of links to various XSLT-related tutorials, tools, etc. The XSLT.com Web site, not surprisingly, is located at www.xslt.com/.

There is a very useful mailing list, called the XSL List, with its main focus on XSLT. Further information, including subscription information, is located at www.mulberrytech.com/xsl/xsl-list/index.html. The list is very active, and the volume of email can be overwhelming at times. A minor irritation is that some email will be bounced erroneously due to supposed HTML contained in it.

SVG Sites

The Scalable Vector Graphics specification is located at www.w3.org/TR/svg. The W3C site also has a useful summary page, which describes key events relating to SVG, product announcements, etc. It includes information about new SVG tools and other information about the progress of SVG and is generally kept up to date.

The SMIL Animation specification that provides the animation elements used in SVG is located at www.w3.org/TR/smil-animation.

There is a useful general SVG mailing list on YahooGroups.com. General information about the SVG-Developers mailing list is available at www.yahoogroups.com/group/svg-developers.

W3C also has an SVG-oriented SVG mailing list. Its focus is more on detail of the SVG specification. To subscribe to it, send an email to www.svg-request@w3.org with SUBSCRIBE in the subject line of the email. Archives of the list can be accessed at http://lists.w3.org/Archives/Public/www-svg/.

XForms Sites

The XForms specification is located at www.w3.org/TR/xforms/. An overview of XForms is located at www.w3.org/MarkUp/Forms/. It includes information about prototype XForms processors.

There is an XForms mailing list on YahooGroups.com for open discussion of XForms. Further information is located at www.yahoogroups.com/group/xforms/. To subscribe to the XForms mailing list, send an email to xforms-subscribe@yahoogroups.com.

W3C has a mailing list dedicated to the discussion of the XForms specification. To subscribe to the list, send an email to www.forms-request@w3.org with SUBSCRIBE in the subject line of the email. Archives of the list are available at http://lists.w3.org/Archives/Public/www-forms/.

XPointer Sites

The XPointer specification is located at www.w3.org/TR/xptr. In addition there is some XPointer-related material, including mention of prototype implementations, on the XLink overview page at www.w3.org/XML/Linking.

There is an XLink mailing list on YahooGroups.com, which covers discussion of XLink and XPointer. Further information is available at www.yahoogroups.com/group/xlink/. To subscribe to the mailing list, send an email to xlink-subscribe@yahoogroups.com.

W3C has a mailing list for comments on the XPointer specification. To subscribe to it, send an email to www.xml-linking-requests@w3.org. Archives of the list are available at http://lists.w3.org/Archives/Public/www-xml-linking-comments.

XQuery Sites

At the present time XQuery sites are not plentiful. Some useful information on XQuery can be located at http://xml.coverpages.org/xmlQuery.html.

The Quip prototype XQuery processor from SoftwareAG.com can be downloaded from www.softwareag.com/developer/downloads/default.htm.

A brief introduction to XQuery is located at www.fatdog.com/XQuery_Intro.html. An evaluation copy of an XQuery Engine may also be downloaded from www.fatdog.com.

An online XQuery demonstration had been available from Microsoft at http://131.107.228.20/, but at the time of writing, the URL seemed to be inaccessible. The alternative URL http://131.107.228.20/xquerydemo/demo.aspx also seemed to be inoperative. When initially operative, it used a non-W3C standard syntax.

There is a mailing list dedicated to XQuery on YahooGroups.com. Further information is located at www.yahoogroups.com/group/XQuery/. To subscribe to the list, send an email to xquery-subscribe@yahoogroups.com.

W3C has an XQuery mailing list that is primarily focused on XQuery, but also allows discussion of other query languages. To join the list, send an email to www.ql-request@w3.org with subscribe in the subject line of the email. Archives of the mailing list are available at http://lists.w3.org/Archives/Public/www-ql/.

XSL-FO Sites

XSL-FO tools can be downloaded from www.antennahouse.com, xml.apache.org and www.x-smiles.org.

An online XSL-FO reference is located at www.zvon.org/xxl/xslfoReference/Output/.

There is an XSL-FO mailing list on YahooGroups.com for open discussion of issues relating to the creation of XSL-FO, including issues relating to XPath. Further information is available at www.yahoogroups.com/group/XSL-FO/. To subscribe to the mailing list, send an email to XSL-FO-subscribe@yahoogroups.com.

W3c has an XSL-FO mailing list. To subscribe, send an email to www-xsl-fo-request@w3.org with SUBSCRIBE in the subject line of the email.

Some discussion of XSL-FO takes place on the XSL mailing list. Further information is located at www.mulberrytech.com/xsl/xsl-list/index.html.

General XML Sites

The Web sites at www.XML.com and www.XMLHack.com provide generally useful articles on XML-related topics. XMLHack.com provides a news-oriented coverage of the XML scene.

Glossary

This glossary defines XPath terms, or terms used with XPath. For some terms, a cross-reference is provided for more extensive discussion of the topic or relevant examples within the book.

NOTE The capitalization of terms in headers may not accurately represent the exact syntax you need when using the term. Check the entry for the correct case to use.

Ancestor Axis One of the 13 XPath axes. The ancestor axis selects the parent of the context node (if it has one), that node's parent, and so on, up to and including the root node. The ancestor axis operates in reverse document order.

Ancestor-or-Self Axis The ancestor-or-self axis includes the ancestor axis as well as the context node. The ancestor-or-self axis operates in reverse document order. *See* **Ancestor axis**.

Attribute A name-value pair that is associated with an XML element and appears in the element's start tag. An attribute is represented by an attribute node in the XPath data model.

Attribute Axis The attribute axis selects all the attributes associated with an element node that is the context node. If the context node is not an element node, then the attribute axis is empty.

Attribute Node The representation in the XPath data model of an XML attribute. For each attribute belonging to an element, an attribute node will be created with the exception of an attribute that is a namespace declaration.

Attribute Set An XSLT term. An attribute set is a named collection of <xsl:attribute> elements.

Attribute Value Template An XSLT term that describes an attribute in an XSLT stylesheet which contains both static and dynamic parts. The dynamic parts are enclosed within curly braces { }. Not all XSLT elements permit attribute value templates.

Axis An axis is one of the 13 directions of traversal of an XPath tree. The nodes reached or selected during traversal depend on the starting point (the "context node") and the axis specified.

Boolean One of the four data types permitted in XPath 1.0. It may take the value "true" or "false."

Built-in Template Rule An implicit template rule, which an XSLT processor will apply in the absence of an explicit template rule that matches a node. Sometimes also called a default template rule.

CDATA Section A sequence of characters contained in an XML document and which is preceded by `<![CDATA[` and followed by `]]>`. Characters such as the "<" character do not require to be escaped when present in a CDATA section.

Character Reference A reference to a character that uses its decimal or hexadecimal Unicode value. Typically, a character reference may be used for characters that cannot readily be typed directly on a keyboard.

Child Axis The child axis is one of the 13 XPath axes and selects the element nodes, comment nodes, processing instruction nodes, and text nodes that are children of an element node or of the root node. The child axis does not include namespace nodes or attribute nodes. The principal element type for the child axis is the element node.

Child Node A child node is a node that is an element, comment, processing instruction, or text node and which has an element node or a root node as its parent.

Closure The property of a language that allows the result of processing to be in a form suitable for processing as the input to further processing. In XSLT the result tree from an XSLT transformation may, when the output is XML, be used as the input to a further XSLT transformation.

Comment A part of XML syntax that allows descriptive or other textual material to be included but not parsed. It begins with the `<!–` character sequence and ends with the `–>` character sequence. A comment is represented in the XPath tree by a comment node.

Comment Node The representation in the XPath tree of a comment in the XML source document. *See* **Comment**.

Context The position in an XPath hierarchy that an XPath processor treats as if it were its current location.

Context Node The node that is the starting point for the traversal of the XPath tree. It may be, but is not always, the same node as the XSLT current node. The XPath location paths "." or "self::node ( )" retrieve the context node.

Context Position An integer that describes a particular place within the context size.

Context Size A part of the XPath context. The context size is the number of nodes in the current node list or the number of nodes selected by a step in an XPath location path. The node returned by the last ( ) function is determined by the context size. *See* **Context**.

Current Node An XSLT term. A node in the XPath tree becomes the current node when it is accessed using <xsl:apply-templates> or <xsl:for-each>. The XSLT current ( ) function accesses the current node. The XSLT current node and the XPath context node will be the same node, except during the processing of a predicate of an XPath location path.

Current Node List An XSLT term. The current node list is an ordered list of nodes in the tree representation of the XML source document. The current node list is deter-

mined by the value of the select attribute of an <xsl:apply-templates> or an <xsl:for-each> element. The ordering within the current node list may be altered from the default (document order) by an <xsl:sort> element.

Current Template Rule An XSLT term relevant when choosing which template is to be activated when the <xsl:apply-imports> element is called.

Descendant Axis One of the 13 XPath axes. The descendant axis recursively selects all the children of the context node, their children, etc.

Descendant-or-Self Axis The descendant-or-self axis contains the members of the descendant axis plus the context node. *See* **Descendant axis**.

Descendants The descendants of a node are the children of that node and the children of the children of that node, and so on.

Document A series of characters is said to be well-formed if the characters conform to the requirements outlined in the XML 1.0 Recommendation. If, in addition, an XML document conforms to the declarations in a document type definition, an XML document is also said to be valid. *See* **Document Type Definition**.

Document Element The document element is the outermost element occurring in an XML document. All other elements in the document are descendants of that element. The node representing the document element is sometimes termed the element root, which is not to be confused with the XPath root node. The node representing the element root is a child element node of the XPath root node.

Document Order The order of the nodes in a node set that corresponds to the order in which the start tags of the elements they represent are present in the XML source document. The ordering of attribute nodes, namespace nodes, and nodes representing separate source documents is ambiguous in the XPath specification.

Document Type Definition A schema that defines the structure of an XML document. A document type definition may exist in two parts: the internal subset, which is enumerated in the Document Type Declaration, and the external subset, which is contained in an external file referenced from the Document Type Declaration.

Element A logical component of an XML document, which possesses a start and end tag and which may possess attributes as name-value pairs. An empty element is permitted to have an empty element tag as an alternative to a start tag followed by an end tag.

Element Node The representation in the XPath in-memory tree of an element in an XML source document. An element node has a parent (either another element node or the root node) and may have children; the allowable children being element nodes, comment nodes, processing instruction nodes, and text nodes, which represent the content of the element in the source document.

Entity A physical unit of information. An entity may be an internal entity or an external entity. Entities may be parsed (which contain XML markup) or unparsed (which contain non-XML, often binary, data). Entities may be general entities (which contain material to be embedded in the source XML document) and parameter entities (which contain material to be included in the Document Type Definition).

Entity Reference A reference to an external or internal entity.

Expanded Name A name related to a QName. The expanded name consists of two parts: the namespace name (also called the namespace URI), which is associated with the namespace prefix of a QName and, secondly, the local part. The precise syntax of an expanded name is not defined in the Namespaces in XML Recommendation.

Expression The most general form of syntactic construct in XPath. An XPath expression may evaluate to one of four data types: a node set, a Boolean, a string, or a number. An expression that returns a node set is termed a location path. An expression may be used in attribute value templates or within the attribute values of several XSLT elements.

Following Axis One of the 13 XPath axes. The following axis selects all nodes that follow the context node in document order but excluding attribute and namespace nodes and also excluding the descendants of the context node.

Following-Sibling Axis The following-sibling axis selects all nodes that have the same parent as the context node and that follow the context node in document order.

Function A logical unit that can be called from within an XPath expression. A function may take arguments and it returns a result. Both XPath and XSLT define core functions. In addition, vendor-specific functions may also be used.

Global Variable An XSLT term that describes a top-level <xsl:variable> element. A global variable is available throughout a stylesheet, except if overridden by a local variable of the same name. *See* **Top-level**.

HyperText Markup Language HTML is a markup language, which is widely used on the World Wide Web and is an application language of SGML. HTML is a common format for output from XSLT transformations.

ID ID is a term used to refer to attributes that have a value unique within an XML document and which are defined as having type ID in an accompanying document type definition. An ID type attribute may be accessed by means of the value of that attribute using the id ( ) function.

Instantiate A term, together with its derivatives "instantiated" and "instantiation," used to refer to the execution of XSLT templates and instructions. The term appears to have been chosen to distinguish the process from the execution of procedural code.

Instruction An XSLT term used to refer to a number of XSLT elements, such as <xsl:variable>, which may occur directly within an <xsl:template> element.

Literal Result Element A literal result element is an element present in an XSLT stylesheet that is not in the XSLT namespace nor is it an XSLT extension element. A literal result element is copied to the output document.

Local Part The part of a QName that follows the namespace prefix and the colon character.

Local Variable A variable defined within an XSLT template.

Location Path An XPath expression that returns a node set. A location path consists of one or more location steps.

Location Step A part or whole of a location path. A location step consists of an axis, a node test, and a predicate.

Mode A mode attribute allows the template rules in an XSLT stylesheet to be compartmentalized, so that a node may be processed more than once.

Named Template An XSLT term for <xsl:template> element that possesses a name attribute. A named template is accessed using the <xsl:call-template> instruction.

Namespace A collection of names that, as defined in the Namespaces in XML Recommendation, is referenced by means of a namespace declaration, which associates a namespace prefix with a namespace URI.

Namespace Axis One of the 13 XPath axes. The namespace axis selects all the namespace nodes associated with the context node. For nodes other than element nodes, the namespace axis is empty.

Namespace Declaration An attribute name-value pair that associates a namespace prefix with a namespace URI.

Namespace Name A synonym of namespace URI. *See* **Namespace URI**.

Namespace Node A node associated in the XPath tree with a parent element node that represents the association of a namespace prefix with a namespace URI. A namespace node is "personal" to its parent node. Any descendant elements nodes for which the same namespace declaration is in scope has its own namespace node to express that relationship.

Namespace Prefix The namespace prefix is part of a QName that acts as a proxy for the namespace URI. A namespace prefix is useful since it can be brief and because some characters allowed in a namespace URI may not be used in an XML name. A namespace prefix must be an NCName.

Namespace URI The Uniform Resource Identifier, which is used to uniquely identify a collection of names in an XML namespace. A synonym of namespace URI is "namespace name."

NaN Not a number. This is a permitted value for a variable whose data type is a number. NaN indicates that the variable value is not a number.

NCName An XML name in which the colon character is not permitted. A namespace prefix is one use of an NCName, since any colon character within a namespace prefix could cause difficulty in parsing the QName, since it would be unclear as to which colon character represented the separator between the namespace prefix and the local part.

Node An object on the XPath tree. XPath 1.0 defines seven node types: root node, element node, attribute node, comment node, namespace node, processing-instruction node, and text node. There is no node to represent an XML declaration or the Document Type Declaration.

Node Set One of the four data types allowed in XPath 1.0. A node set is an unordered set of nodes without duplicates.

Node Test Part of a location step in an XPath location path. The other parts are the axis and the optional predicate.

Number One of the four permitted data types in XPath.

Parent A node that is one level higher in the tree hierarchy than the context node. Only a root node or an element node may be a parent node.

Parse Tree An alternative name for the hierarchical in-memory representation of an XML document.

Pattern A pattern may occur on the <xsl:template>, <xsl:key>, and <xsl:number> elements and is a subset of XPath expressions that a node may or may not match.

Preceding Axis One of the 13 XPath axes. The preceding axis selects, in reverse document order, all the nodes that precede the context node with the exception of the node's ancestors and any attribute nodes or namespace nodes.

Preceding-Sibling Axis The preceding-sibling axis selects all nodes that have the same parent node as the context node and come before the context node in document order.

Predicate Part of an XPath expression or location path that filters the nodes selected in a location step.

Priority A floating point number that represents the priority of an XSLT template rule.

Processing Instruction A construct in XML syntax intended to convey information or instructions to a target application. A processing instruction has the "<?" character sequence as its initial delimiter and the "?>" character sequence as its terminating delimiter.

Processing Instruction Node A node in an XPath tree that represents a processing instruction in the source XML document.

QName A Qualified Name. A QName consists of an optional namespace prefix, an optional colon character (required if there is a namespace prefix), and a local part.

Result Tree The hierarchical output from an XSLT processor. The result tree may, at a subsequent stage, be serialized as an XML document or may be used as the input to a further XSLT transformation. The latter use of the result tree depends on the property of closure, which XSLT exemplifies.

Result Tree Fragment An XSLT term referring to the data type of a temporary in-memory tree created by instantiating a nonempty <xsl:variable> element.

Root Node The XPath 1.0 representation of the document entity of an XML document. The root node serves as the top of the hierarchy in a parse tree. The node representing the element root (document element) is a child of the root node.

Serialized The form in which we usually view XML documents, where one character follows another in a series. It contrasts with the in-memory tree representation of an XML document.

Source Document The XML document to which an XSLT transformation is applied.

Step A synonym for location step. Also used to refer to an XPointer step. *See* **Location step**.

String One of the four data types permitted in XPath 1.0. A string is a sequence of zero or more Unicode characters.

String-value Each node in an XPath tree has a string-value. The method for the calculation of the string-value on each of XPath 1.0's seven node types is described in Chapter 3.

Stylesheet An XSLT stylesheet. In the simplest case that is an <xsl:stylesheet> element and its content. An XSLT stylesheet may refer to an <xsl:stylesheet>, its content, and stylesheet modules referenced using the <xsl:import> or <xsl:include> elements.

Template An XSLT term referring to the content of an <xsl:template> element.

Template Rule An XSLT term that refers to an <xsl:stylesheet> element that possesses a match attribute.

Text Node A node in an XPath tree representing character data within an XML document. When the XPath tree is constructed, the number of text nodes is minimized by merging any adjacent text into a single node. The content of a CDATA section will form all or part of a text node. Character and entity references occurring in text are expanded to their replacement text and then inserted into a text node.

Top-level An XSLT term to describe elements that are children of an <xsl:stylesheet> or <xsl:transform> element. The term is a little misleading since these "top-level" elements are actually second-level elements.

Tree An abstract data structure created in memory that represents the logical content of an XML document. An XPath tree has one and only one root node. A tree need not itself represent a well-formed XML document, since the tree representation of an external entity may be intended for insertion into another XML document, which itself possesses a single element root.

Unparsed Entity An unparsed entity is an entity declared in a Document Type Definition with an associated notation, which indicates that the content of the entity is not XML.

URI Uniform Resource Identifier. An address that provides an unambiguous location for a resource. The term URI includes the more familiar URL (Uniform Resource Locator) as well as the URN (Uniform Resource Name).

Variable Binding The associating of a variable, such as in an <xsl:variable> element, with the value of that variable.

Variable Reference A reference within an XPath expression or location path to a variable. The reference to a variable takes the form $VariableName.

Well-formed A term to describe a document that satisfies the requirements of the XML 1.0 Recommendation with respect to syntax and structure.

Whitespace An XML 1.0 Recommendation term for one or more successive characters of space, newline, carriage return, and tab.

XPath The XML Path Language.

TEAMFLY

Index

: (colon), 19
& (ampersand), 9
‘ (apostrophe), 9
{ } (curly braces), 38
> (greater than), 9
< (less than), 9
“ “ (quote), 9

A

Abbreviated absolute syntax, 78–80, 211–216
  and ancestor axis, 214
  and ancestor-or-self axis, 216
  attribute axis in, 212, 213
  child axis in, 211–212
  descendant axis in, 212–215
  and descendant-or-self axis, 215
  and following axis, 215
  and following-sibling axis, 214
  and namespace axis, 215
  and parent axis, 214
  and preceding axis, 215
  and preceding-sibling axis, 215
  and self axis, 216
Abbreviated relative syntax, 78, 80, 216–221
  and ancestor axis, 220
  and ancestor-or-self axis, 221
  attribute axis in, 218–219
  child axis in, 216–217
  descendant axis in, 219, 220
  and descendant-or-self axis, 221
  and following axis, 220
  and following-sibling axis, 220
  and namespace axis, 221
  in parent axis, 219
  and preceding axis, 221
  and preceding-sibling axis, 220
  and self axis, 221
Abbreviated syntax, 210
Abstract data set, 155
Access control, 385
Accessors, 485–486
Adobe SVG Viewer, 310
Ampersand (&) character, 9
Ancestor axis, 87–88
  and abbreviated absolute syntax, 214
  and abbreviated relative syntax, 220
  definition of, 493
  in unabbreviated absolute syntax, 166, 178
  in unabbreviated relative syntax, 190–193
Ancestor-or-self axis, 98
  and abbreviated absolute syntax, 216
  and abbreviated relative syntax, 221

Ancestor-or-self axis (*cont.*)
definition of, 493
in unabbreviated absolute syntax, 166, 181
in unabbreviated relative syntax, 207–209
Animated bar charts, 319–323
AnyURI datatype, 481
Apache Organization, 52
Apostrophe (') character, 9
Apply imports element, 36
Apply templates element, 36–37
Asymmetric key encryption, 403
Attributes:
creating, 280–282
definition of, 493
selecting elements by presence of, 425–431
selecting elements by value of, 430–435
Attributes element, 37–38
Attribute axis, 95, 96
in abbreviated absolute syntax, 212, 213
in abbreviated relative syntax, 218–219
definition of, 493
in unabbreviated absolute syntax, 166, 169–173
in unabbreviated relative syntax, 186–188
Attribute information items, 159–160
Attribute nodes, 64, 149–150, 398, 493
Attribute set, 30, 493
Attribute specifications, 5–7
Attribute value templates, 48–49, 493
Authentication, 385
Authorization, 385
Average() function (XForms), 381
Axis, 57, 82–98
ancestor axis, 87–88, 493
ancestor-or-self axis, 98, 493
attribute axis, 95, 96, 493
child axis, 83–85, 494
definition of, 494
descendant axis, 84–87, 495
descendant-or-self axis, 98, 495
following axis, 91–94, 496
following-sibling axis, 88–90, 496
namespace axis, 96–98, 497
parent axis, 87
preceding axis, 93–96, 497
preceding-sibling axis, 90, 497
self axis, 98

B

Bar charts:
animated, 319–323
static, 310–319
Bare names (XPointer), 344, 345
Base64Binary datatype, 481
Binding, variable, 499
Binding element (XForms), 371
Binding expressions (XForms), 372, 377–378
Binding (XForms), 371
Boolean datatype, 481, 494
Boolean functions, 135, 245–248, 475–478
boolean() function, 135, 245–246, 475–477
false() function, 135, 246–247
lang() function, 135, 247–248
not() function, 135, 248, 477–478
true() function, 135, 248
in XForms, 382
Boolean values, 44, 48
Built-in template rule, 494
Business information, reusing, 277–280
Button form control (XForms), 362

C

Calculate property (XForms model), 369
Call template element, 38–39
Canonical binding expressions (XForms), 378
Canonicalization (c14n), 393–394, 400–402
Canonical XML, 64, 383, 386–402, 404
canonicalization process in, 393, 400–402
with digital signatures, 404
document order in, 397
document subsets in, 399

need for, 386–392
octet stream, conversion of node-set into, 397–399
purpose of, 392–393
role of XPath in, 394–399
and well-formed documents, 399–400
Cardiff.com, 375
Case conversion function, 133–134, 245
CDATA sections, 11–12, 151, 494
Ceiling() function, 115, 238
Character data, 9, 11–12. *See also* String functions; Text
Character information items, 161
Character-point (XPointer), 342
Character references, 7–8, 494
Child axis, 83–85
in abbreviated absolute syntax, 211–212
in abbreviated relative syntax, 216–217
definition of, 494
in unabbreviated absolute syntax, 167–170
in unabbreviated relative syntax, 183–186
Child elements (XForms), 363
Child node, 494
Child sequences (XPointer), 344–346
Choose element, 39–40, 48
Closure, 494
Collapsed range (XPointer), 343
Colon character, 19
Comment, 11, 494
Comment element, 40
Comment information items, 161
Comment nodes, 64, 150, 399, 494
Computed expressions (XForms), 372
Concatenate string function, 120–122, 239
Concat() function, 120–122, 239–241, 466–468
Confidentiality, 385
Container node (XPointer), 342
Containing document (XForms), 372
Contains() function, 122–123, 241–243, 468–473
Context, 494
Context node, 57, 65–71, 135, 234, 247–248, 248–249, 494
Context position, 237, 494
Context size, 231–232, 494
Copy element, 40–41
Copy-of element, 41
Count() function, 104–105, 224–230, 441–445
Count-non-empty() function (XForms), 381
Covering range (XPointer), 343–344
Curly braces ({ }), 38, 48–49
Currency datatype (XForms), 380
Current() function, 248–249
Current node, 71, 72, 248–249, 494
Current node list, 494–495
Current template rule, 495

D

Data integrity, 385
Data models:
XPath, *see* XPath data model
XPointer, 341–344
Datatypes (XForms), 370–372, 378–380
Date datatype, 481
DateTime datatype, 481
Decimal datatype, 481
Decimal format element, 30
Declarations:
entity, 7, 8
XML, 10
Descendants, 495
Descendant axis, 84–87
in abbreviated absolute syntax, 212–215
in abbreviated relative syntax, 219, 220
definition of, 495
in unabbreviated absolute syntax, 174–178
in unabbreviated relative syntax, 187–191
Descendant-or-self axis, 98
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 221
definition of, 495

Descendant-or-self axis (*cont.*)
in unabbreviated absolute syntax, 179–180
in unabbreviated relative syntax, 203–206
Digital signatures, 404
DOCTYPE declaration, 394–395, 398
Documents, 2–14
definition of, 495
and in-memory trees, 145–146
logical structure of, 4–7
normalization of attributes, 13–14
output, 62, 63
physical structure of, 7–10
source, 498
syntax of, 10–12
valid vs. non-valid, 12–13
well-formed, 12
Document element, 36, 148, 495
Document entity, 7
Document() function, 249–252
Document information item, 157–158
Document Object Model (DOM), 152–155
Document order, 495
Document Type Declaration, 10, 155
Document type declaration information item, 162
Document type definition, 495
Document Type Definition (DTD), 5, 10
DOM, *see* Document Object Model
Double datatype, 481
Double quote character, 9
DTD, *see* Document Type Definition
Duration datatype, 481

E

ECMAScript, 354
Element(s), 4–7
definition of, 495
document, 148, 495
selection of, 407–440
by attribute presence, 425–431
by attribute value, 430–435
following elements, 423–425
by name, 408–409
by parent characteristics, 410
by position, 414–418
preceding elements, 419–423
by value, 410–414
when passing parameters, 435–440
Element-available() function, 251, 252
Element element, 41–42
Element information items, 158–159
Element node, 63–64, 148–149, 398, 495
Element root, 148
Element type name, 5
Empty elements, 5
End-point() function (XPointer), 349
End tag, 4–5, 9
Entity, 7–9, 495
Entity declarations, 7, 8
Entity reference, 495
Epilog, 148
Error messages:
stylesheet, 50
unknown host, 112
Expanded name, 149, 495
Expression, 64–74, 75. *See also* Location paths
definition of, 496
function calls in, 72–74
Extensible Access Control Markup Language (XACML), 385, 405–406
Extensible Markup Language (XML), 1–22
documents in, 2–14
logical structure of, 4–7
normalization of attributes, 13–14
physical structure of, 7–10
syntax of, 10–12
valid vs. non-valid documents, 12–13
well-formed document, 12
getting help with, 24
as meta-language, 2
namespaces in, 19–22
Recommendations for, 2, 17–18
required knowledge of, 23
restructuring, *see* Restructuring of XML
software for working with, 23–24
support site for, 53–54
uses of, 14–16

Web sites, 492
working with, 18–19
Extensible Stylesheet Language Transformations (XSLT), 24–54, 248–259. *See also specific topics*
attribute value templates in, 48–49
functions in, 248–259
current() function, 248–249
document() function, 249–252
element-available() function, 251, 252
format-number() function, 252–253
function-available() function, 253
generate-id() function, 253–256
key() function, 256–258
system-property() function, 259
unparsed-entity-uri() function, 259
processors for, 49–54
Microsoft MSXML3, 53–54
Oracle XML Development Kit, 54
Saxon/Instant Saxon, 49–51
Xalan, 52–53
stylesheets in, 24–26
top-level elements, 29–36
xsl:namespace-alias element, 31–32
<xsl:attribute-set> element, 30
<xsl:decimal-format element, 30–31
<xsl:import> element, 30
<xsl:include> element, 31
<xsl:key> element, 31
<xsl:output> element, 32–33
<xsl:param> element, 33–34
<xsl:preserve-space> element, 34
<xsl:strip-space> element, 34
<xsl:template> element, 34–35
<xsl:variable> element, 35–36
Web sites, 490
XPath in, 28
<xsl:apply-imports> element, 36
<xsl:apply-templates> element, 36–37
<xsl:attribute> element, 37–38
<xsl:call-template> element, 38–39
<xsl:choose> element, 39–40
<xsl:comment> element, 40
<xsl:copy> element, 40–41
<xsl:copy-of> element, 41
<xsl:element> element, 41–42
<xsl:fallback> element, 42–43
<xsl:for-each> element, 43
<xsl:if> element, 43–46
<xsl:number> element, 46
<xsl:otherwise> element, 47
<xsl:processing-instruction> element, 47
<xsl:sort> element, 47
<xsl:stylesheet> element, 28–29
<xsl:text> element, 48
<xsl:value-of> element, 48
<xsl:when> element, 48
<xsl:with-param> element, 48

F

Facets (XForms), 372
Fallback element, 42–43
False() function, 118, 135, 246–247
False value, 44
Float datatype, 481
Floor() function, 116, 238–239
Following axis, 91–94
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 220
definition of, 496
in unabbreviated absolute syntax, 166, 179
in unabbreviated relative syntax, 198–200
Following-sibling axis, 88–90
and abbreviated absolute syntax, 214
and abbreviated relative syntax, 220
definition of, 496
in unabbreviated absolute syntax, 166, 178
in unabbreviated relative syntax, 193–196
For-each element, 43
Format-number() function, 252–253
Form control (XForms), 372
Forms, creating, *see* XForms
Forward axis, 100–101
Fragment, result tree, 498
Full XPointers, 344, 345
Function-available() function, 253
Function calls to, 72–74

Functions, 104–135, 223–259, 441–478
Boolean, 135, 245–248, 475–478
boolean() function, 135, 245–246, 475–477
false() function, 135, 246–247
lang() function, 135, 247–248
not() function, 135, 248, 477–478
true() function, 135, 248
definition of, 496
node set, 104–114, 224–237, 441–461
count() function, 104–105, 224–230, 441–445
id() function, 105–108, 230–231, 446–448
last() function, 108–109, 231–232, 449–452
local-name() function, 109–111, 232–234, 451–453
name() function, 111–113, 234–235, 454–457
namespace-uri() function, 112, 113, 235–236, 456–459
position() function, 113–114, 237, 458–461
number, 115–120, 238–239, 460–466
ceiling() function, 115, 238
false() function, 118
floor() function, 116, 238–239
number() function, 117–118, 239
round() function, 118–119, 239
sum() function, 120, 239, 460–466
reasons for, 223–224
string, 120–134, 239–245, 466–474
concat() function, 120–122, 239–241, 466–468
contains() function, 122–123, 241–243, 468–473
normalize-space() function, 123–125, 243, 244
starts-with() function, 124–126, 244, 472–474
string() function, 126–128, 244
string-length() function, 127–129, 244
substring-after() function, 130–132, 245
substring-before() function, 131–133, 245
substring() function, 129–130, 245
translate() function, 133–134, 245
XForms, 381–382
XSLT, 248–259
current() function, 248–249
document() function, 249–252
element-available() function, 251, 252
format-number() function, 252–253
function-available() function, 253
generate-id() function, 253–256
key() function, 256–258
system-property() function, 259
unparsed-entity-uri() function, 259

G

GDay datatype, 481
Generate-id() function, 253–256
Global parameters, 33–34
Global variable, 496
GMonth datatype, 481
GMonthDay datatype, 481
Greater than character, 9
GYear datatype, 481
GYearMonth datatype, 481

H

Help, XML, 24, 53–54
Here() function (XPointer), 349
HexBinary datatype, 481
HyperText Markup Language (HTML), 10, 496
forms in, 353–355
producing, with XPath/XSLT, 283–304
lists, 285–292
pseudo schema, 299–304
tables, 293–298

I

IDs, 253–256, 496
ID attribute, 230, 380–381
Id() function, 105–108, 230–231, 446–448
IETF (Internet Engineering Task Force), 384

If element, 40, 43–46, 253
Immediately enclosing element (XForms), 372
Immediate-recalculate property (XForms), 374
Immediate-refresh property (XForms), 373
Immediate-revalidate property (XForms), 374
Import element, 30
Importing stylesheets, 36
Include element, 31
Index (XPointer), 342
Information items, 155
Information items (XML Information Set), 156–163
  attribute information items, 159–160
  character information items, 161
  comment information items, 161
  document information item, 157–158
  document type declaration information item, 162
  element information items, 158–159
  namespace information items, 163
  notation information items, 163
  processing instruction information items, 160
  unexpanded entity reference information items, 160–161
  unparsed entity information items, 162–163
In-memory trees, 138–152
  and documents, 145–146
  nodes of, 146, 148–152
    attribute nodes, 149–150
    comment nodes, 150
    element nodes, 148–149
    namespace nodes, 150–151
    processing instruction nodes, 151
    root node, 148
    test nodes, 151
  result tree, 144–146
  source tree, 138–143, 147
Instance data item (XForms), 372
Instance data (XForms), 372, 375–376
Instantiate, 496
Instant Saxon, 49–51
Instruction:
  definition of, 496
  processing, 11, 498
Integer:
  largest, 116, 238–239
  nearest, 118–119, 239
  smallest, 115, 238
International Organization for Standardization (ISO), 7
Internet Engineering Task Force (IETF), 384
ISO/IEC 10646 character set, 7
ISO (International Organization for Standardization), 7

J

Java Development Kit (JDK), 49
Java Virtual Machine (JVM), 50

K

Kay, Michael, 49
Keys, 403
Key element, 31
Key() function, 256–258

L

Lang() function, 135, 247–248
Language identification, 12
Largest integer, 116, 238–239
Last() function, 108–109, 231–232, 449–452
Length of string function, 127–129
Less than character, 9
Lexical space (XForms), 372
Line charts, static, 323–327
<line> element (SVG), 306–308
LiquidOffice, 375
Lists, creating HTML, 285–292
Literal result element, 496
Local-name() function, 109–111, 232–234, 451–453
Local part, 496
Local variable, 496
Location paths, 57, 75–82
  definition of, 496

Location paths (*cont.*)
describing, 75–80
abbreviated absolute syntax, 78–80
abbreviated relative syntax, 78, 80
unabbreviated absolute syntax, 75–80
unabbreviated relative syntax, 77, 79
locations steps in, 80–82
axes as part of, 82–98
node tests as part of, 98–100
predicates as part of, 100–104
Location step, 496
Locations (XPointer), 339
Logical structure (documents), 4–7
Logical tokens, 60

M

Main templates, 72
Match attribute, 77
Matching, using XPath for, 22
Max() function (XForms), 381
Meta-language, XML as, 2
Microsoft MSMXL, 53–54
Microsoft XSL, 29
Min() function (XForms), 381
Mode, 496
Model item property (XForms), 372
Model item (XForms), 372
Monetary datatype (XForms), 380
Mozquito.com, 375
MSXML, 53–54

N

Name(s):
element selection by, 408–409
node, 152, 234–235
Named template, 496
Name() function, 111–113, 234–235, 454–457
Name property (XForms model), 369
Namespace, 496
Namespace-alias element, 31–32
Namespace axis, 96–98
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 221
definition of, 497
in unabbreviated absolute syntax, 166, 179
in unabbreviated relative syntax, 201–203
Namespace declaration, 497
Namespace information items, 163
Namespace name, 497
Namespace node, 64, 150–151, 154, 398, 497
Namespace prefix, 497
Namespaces, 19–22
Namespace URI, 497
Namespace-uri() function, 112, 113, 235–236, 456–459
NaN, 497
NCName, 19, 497
Nearest integer, 118–119, 239
Nesting of elements, 62
Nodes, 57, 60–72
context, 65–71, 494
current, 71, 72, 494
definition of, 60–63, 497
in-memory trees, 146, 148–152
attribute nodes, 149–150, 493
comment nodes, 150
element nodes, 148–149
namespace nodes, 150–151
processing instruction nodes, 151
root node, 148
text nodes, 151
types of, 63–64
Node names, 152, 234–235
Node-point (XPointer), 342
Node position, 195
Node set, 497
Node-set functions, 104–114, 224–237, 441–461
count() function, 104–105, 224–230, 441–445
id() function, 105–108, 230–231, 446–448
last() function, 108–109, 231–232, 449–452
local-name() function, 109–111, 232–234, 451–453

name() function, 111–113, 234–235, 454–457
namespace-uri() function, 112, 113, 235–236, 456–459
position() function, 113–114, 237, 458–461
Node test, 98–100, 497
NodeValue method, 153
Nonrepudiation, 385
Non-valid documents, 12–13, 231
Normalization, 13–14
Normalize-space() function, 123–125, 243, 244
NOTATION datatype, 481
Notation information items, 163
Notations, 9–10
Not() function, 135, 248, 477–478
Now() function (XForms), 382
Number:
definition of, 497
format, 252–253
Number element, 46
Number() function, 117–118, 239
Number functions, 115–120, 238–239, 331, 460–466
ceiling() function, 115, 238
false() function, 118
floor() function, 116, 238–239
number() function, 117–118, 239
round() function, 118–119, 239
sum() function, 120, 239, 460–466
in XForms, 381

O

OASIS (Organization for Advancement of Structured Information Standards), 384
Oracle XML Development Kit, 54
Order, attribute, 6
Organization for Advancement of Structured Information Standards (OASIS), 384
Origin() function (XPointer), 349
Otherwise element, 47
Output documents, 62
Output element, 32–33
Output form control (XForms), 362
Output trees, 63

P

Param element, 33–34
Parameters:
global, 33–34
values of, 48
Parameter entities, 9
Parent, 497
Parent axis, 87
and abbreviated absolute syntax, 214
in abbreviated relative syntax, 219
in unabbreviated absolute syntax, 166, 178
in unabbreviated relative syntax, 206–208
Parent characteristics, element selection by, 410
Parsed entities, 7–8
Parse tree, 497
Paths, location, *see* Location paths
Pattern, 497
Perl, 379
Point (XPointer), 340, 342–343
Position, element selection by, 414–418
Position, node, 195
Position() function, 72–74, 113–114, 237, 458–461
Preceding axis, 93–96
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 221
definition of, 497
in unabbreviated absolute syntax, 166, 179
in unabbreviated relative syntax, 199–201
Preceding-sibling axis, 90
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 220
definition of, 497
in unabbreviated absolute syntax, 166, 178
in unabbreviated relative syntax, 196–198
Predefined entities, 9

Predicates, 100–104, 498
Prefix, namespace, 497
Preserve-space element, 34
Priority, 498
Priority property (XForms model), 370
Private keys, 403
Processing instruction, 11, 47, 151, 498
Processing-instruction element, 47
Processing instruction information items, 160
Processing instruction node, 64, 151, 399, 498
Proximity position, 101
Public keys, 403

Q

Qname, 19–20, 498
QName datatype, 481
Qualified names, 19
Quotes, as delimeters, 6
Quote character, 9

R

Range form control (XForms), 362
Range() function (XPointer), 349
Range-inside() function (XPointer), 350
Range-to() function (XPointer), 350
Range (XPointer), 340, 343–344
Read-only properties (XForms), 374
ReadOnly property (XForms model), 369
Reasons for functions, 223–224
Recommendations, XML, 2, 17–18
<rect> element (SVG), 308–309
Relevant property (XForms model), 369
Required property (XForms model), 369
Reset form control (XForms), 363
Reset() function (XForms), 382
Restructuring of XML, 261–282
  attributes, creating, 280–282
  and limitations of XPath, 261–262
  reordering of content, 274–277
  and reuse of business information, 277–280
  and selection/creation of elements, 267–274
    with <xsl:copy> element, 270–272
    with <xsl:copy-of> element, 273–274
    with <xsl:element> element, 267–270
    with <xsl:output> element, 262–267
Result documents, 62, 63, 144–146
Result tree, 63, 144–146, 498
Result tree fragment, 498
Reverse axis, 100–101
Root, element, 148
Root node, 61–63, 148, 166, 398, 498
Round down function, 116
Round() function, 118–119, 239
Round up function, 115

S

SAML, *see* Security Language Markup Language
Saxon, 49–51
Scalable Vector Graphics (SVG), 266, 305–310
  creating, with XPath/XSLT, 305, 310–332
    animated bar charts, 319–323
    limitations of, 331–332
    static bar chart, 310–319
    static line charts, 323–327
    weather charts in scrolling text windows, 327–331
  features of, 306
  <line> element in, 306–308
  precursors to, 306
  <rect> element in, 308–309
  <text> element in, 309
  viewing, 310
  Web sites, 490–491
Schema, 5
Schemes (XPointer), 347–348
Script element, 30
Secret form control (XForms), 360
Security, XML, 383–406
  canonical XML for, *see* Canonical XML
  principles of, 384–385
  and XACML, 405–406
  XML signatures for, 402–405
Security Language Markup Language (SAML), 384

TEAMFLY

SelectBoolean form control (XForms), 361–362
SelectMany form control (XForms), 361
SelectOne form control (XForms), 360–361
Self axis, 98
  and abbreviated absolute syntax, 216
  and abbreviated relative syntax, 221
  in unabbreviated absolute syntax, 181
  in unabbreviated relative syntax, 209–210
Serialization, 32
Serialized files, 60, 498
SGML, *see* Standard Generalized Markup Language
Signatures, XML, 402–405
Smallest integer, 115, 238
Sort element, 47
Source documents, 145–146, 249–252, 498
Source trees, 62, 138–143, 147
Space(s), 34, 123–125, 243, 244
Standard Generalized Markup Language (SGML), 14
Start-point() function (XPointer), 350
Starts-with() function, 124–126, 244, 472–474
Start tag, 4–5, 9
Static bar charts, 310–319
Static line charts, 323–327
Step, 498
String, 498
String datatype, 481
String() function, 126–128, 244
String functions, 120–134, 239–245, 466–474
  concat() function, 120–122, 239–241, 466–468
  contains() function, 122–123, 241–243, 468–473
  normalize-space() function, 123–125, 243, 244
  starts-with() function, 124–126, 244, 472–474
  string() function, 126–128, 244
  string-length() function, 127–129, 244
  substring-after() function, 130–132, 245
  substring-before() function, 131–133, 245
  substring() function, 129–130, 245
  translate() function, 133–134, 245
  in XForms, 382
String-length() function, 127–129, 244
String-range() function (XPointer), 350
String-value, 498
Strip-space element, 34, 244
Stylesheets, 24–26, 28–29, 498
Stylesheets, importing, 36
Submit form control (XForms), 362–363
Submit() function (XForms), 382
Sub-resource (XPointer), 340–341
Substring-after() function, 130–132, 245
Substring-before() function, 131–133, 245
Substring() function, 129–130, 245
Sum() function, 120, 239, 460–466
SVG, *see* Scalable Vector Graphics
Syntax, 57–60, 165–221
  abbreviated absolute, 211–216
    and ancestor axis, 214
    and ancestor-or-self axis, 216
    attribute axis in, 212, 213
    child axis in, 211–212
    descendant axis in, 212–215
    and descendant-or-self axis, 215
    and following axis, 215
    and following-sibling axis, 214
    and namespace axis, 215
    and parent axis, 214
    and preceding axis, 215
    and preceding-sibling axis, 215
    and self axis, 216
  abbreviated forms of, 210
  abbreviated relative
    and ancestor axis, 220
    and ancestor-or-self axis, 221
    attribute axis in, 218–219
    child axis in, 216–217
    descendant axis in, 219, 220
    and descendant-or-self axis, 221
    and following axis, 220
    and following-sibling axis, 220

Syntax, abbreviated relative (*cont.*)
and namespace axis, 221
in parent axis, 219
and preceding axis, 221
and preceding-sibling axis, 220
and self axis, 221
ancestor axis
and abbreviated absolute syntax, 214
and abbreviated relative syntax, 220
in unabbreviated absolute syntax, 178
in unabbreviated relative syntax, 190–193
ancestor-or-self axis
and abbreviated absolute syntax, 216
and abbreviated relative syntax, 221
in unabbreviated absolute syntax, 181
in unabbreviated relative syntax, 207–209
attribute axis
in abbreviated absolute syntax, 212, 213
in abbreviated relative syntax, 218–219
in unabbreviated absolute syntax, 169–173
in unabbreviated relative syntax, 186–188
child axis
in abbreviated absolute syntax, 211–212
in abbreviated relative syntax, 216–217
in unabbreviated absolute syntax, 167–170
in unabbreviated relative syntax, 183–186
descendant axis
in abbreviated absolute syntax, 212–215
in abbreviated relative syntax, 219, 220
in unabbreviated absolute syntax, 174–178
in unabbreviated relative syntax, 187–191
descendant-or-self axis
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 221
in unabbreviated absolute syntax, 179–180
in unabbreviated relative syntax, 203–206
document, 10–12
following axis
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 220
in unabbreviated absolute syntax, 179
in unabbreviated relative syntax, 198–200
following-sibling axis
and abbreviated absolute syntax, 214
and abbreviated relative syntax, 220
in unabbreviated absolute syntax, 178
in unabbreviated relative syntax, 193–196
namespace axis
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 221
in unabbreviated absolute syntax, 179
in unabbreviated relative syntax, 201–203
parent axis
and abbreviated absolute syntax, 214
in abbreviated relative syntax, 219
in unabbreviated absolute syntax, 178
in unabbreviated relative syntax, 206–208
preceding axis
and abbreviated absolute syntax, 215

and abbreviated relative syntax, 221
in unabbreviated absolute syntax, 179
in unabbreviated relative syntax, 199–201
preceding-sibling axis
and abbreviated absolute syntax, 215
and abbreviated relative syntax, 220
in unabbreviated absolute syntax, 178
in unabbreviated relative syntax, 196–198
self axis
and abbreviated absolute syntax, 216
and abbreviated relative syntax, 221
in unabbreviated absolute syntax, 181
in unabbreviated relative syntax, 209–210
unabbreviated absolute, 166–181
ancestor axis in, 178
ancestor-or-self axis in, 181
attribute axis in, 169–173
child axis in, 167–170
descendant axis in, 174–178
descendant-or-self axis in, 179–180
following axis in, 179
following-sibling axis in, 178
namespace axis in, 179
parent axis in, 178
preceding axis in, 179
preceding-sibling axis in, 178
self axis in, 181
unabbreviated relative, 181–210
ancestor axis in, 190–193
ancestor-or-self axis in, 207–209
attribute axis in, 186–188
child axis in, 183–186
descendant axis in, 187–191
descendant-or-self axis in, 203–206
following axis in, 198–200
following-sibling axis in, 193–196
namespace axis in, 201–203
parent axis in, 206–208
preceding axis in, 199–201
preceding-sibling axis in, 196–198
self axis in, 209–210
System-property() function, 259

T

Tables, creating HTML, 293–298
Templates, 36–37
attribute value, 38, 48–49, 493
call, 38–39
definition of, 498
main, 72
and parameter values, 48
Template element, 34–35
Template rule, 498
built-in, 494
current, 495
Tests, node, 98–100, 497
Textbox form control (XForms), 359–360
Text declarations, 8
Text element, 48
<text> element (SVG), 309
Text nodes, 64, 151, 153–154, 398–399, 498
Time datatype, 481
Top-level, 29–36, 498
Transformation sheets, 24, 28
Translate() function, 133–134, 245
Trees:
definition of, 499
hierarchical, 61–63
in-memory, *see* In-memory trees
True() function, 135, 248
True value, 44
Types, element, 5
Type property (XForms model), 369

U

Unabbreviated absolute syntax, 75–80, 166–181
ancestor axis in, 166, 178
ancestor-or-self axis in, 166, 181
attribute axis in, 166, 169–173
child axis in, 167–170
descendant axis in, 174–178
descendant-or-self axis in, 179–180

Unabbreviated absolute syntax (*cont.*)
following axis in, 166, 179
following-sibling axis in, 166, 178
namespace axis in, 166, 179
parent axis in, 166, 178
preceding axis in, 166, 179
preceding-sibling axis in, 166, 178
self axis in, 181
Unabbreviated relative syntax, 77, 79, 181–210
ancestor axis in, 190–193
ancestor-or-self axis in, 207–209
attribute axis in, 186–188
child axis in, 183–186
descendant axis in, 187–191
descendant-or-self axis in, 203–206
following axis in, 198–200
following-sibling axis in, 193–196
namespace axis in, 201–203
parent axis in, 206–208
preceding axis in, 199–201
preceding-sibling axis in, 196–198
self axis in, 209–210
Unexpanded entity reference information items, 160–161
Uniform Resource Identifier (URI), 20, 21
definition of, 499
namespace, 497
namespace function, 112, 113, 235–236
unparsed entity, 259
Unparsed entity, 7, 499
Unparsed entity information items, 162–163
Unparsed-entity-uri() function, 259
UploadMedia form control (XForms), 360
URI, *see* Uniform Resource Identifier
Use-nils property (XForms), 374

V

Validate property (XForms model), 370
Valid documents, 12–13, 231
Value, element selection by, 410–414
Value-of element, 48
Value space (XForms), 372
Variable binding, 499
Variable element, 35–36
Variable reference, 499
Variables, assignment of, 36
Version attribute, 10

W

W3C, *see* World Wide Web Consortium
Weather chart, 327–331
Web sites, 489–492
general XML, 492
SVG, 490–491
W3C, 489–490
XForms, 491
XPath, 490
XPointer, 491
XQuery, 491–492
XSL-FO, 492
XSLT, 490
Well-formed documents, 12, 156, 399–400, 499
When element, 40, 48
Whitespace, 34, 123–125, 243, 244
definition of, 499
in text elements, 48
WinZip, 51
Wireless Markup Language (WML), 357, 380
With-param element, 48
WML, *see* Wireless Markup Language
World Wide Web Consortium (W3C), 17–18, 489–490
and Canonical XML, 64
and Document Object Model, 152–155

X

XACML, *see* Extensible Access Control Markup Language
Xalan, 52–53
<xform:bind> element, 373
<xform:instance> element, 373
<xform:model> element, 373
XForms, 353–382
Boolean functions, 382
containing documents, multiple forms in, 380–381
converting XHTML forms to, 363–366
elements of, 372–373

form controls in, 358–363
button form control, 362
and child elements, 363
output form control, 362
range form control, 362
reset form control, 363
secret form control, 360
selectBoolean form control, 361–362
selectMany form control, 361
selectOne form control, 360–361
submit form control, 362–363
textbox form control, 359–360
uploadMedia form control, 360
functions in, 381–382
HTML/XHTML forms vs., 355
model for, 368–371
item properties in, 369–370
using datatypes in, 370–371
multiple forms in containing documents, 380–381
number functions, 381
processors for, 374–375
properties of, 373–374
purpose of, 354
purpose vs. presentation in, 357–358
string functions, 382
terminology used with, 371–372
user interface, 355–357, 366–368
dynamic interface, 366–367
interface templates, 368
layout, 368
repeating items, 367–368
Web sites, 491
XPath in, 375–380
binding expressions, 377–378
datatypes, 378–380
instance data, 375–376
non-outermost binding elements, context for, 377
outermost binding elements, context for, 376–377
Xforms-property() function (XForms), 382
<xform:submitInfo> element, 373
<xform:xform> element, 372
XHTML, forms in, 355, 363–366
XML, *see* Extensible Markup Language
XML declarations, 10
XML Information Set, 155–163
attribute information items, 159–160
character information items, 161
comment information items, 161
document information item, 157–158
document type declaration information item, 162
element information items, 158–159
namespace information items, 163
notation information items, 163
processing instruction information items, 160
unexpanded entity reference information items, 160–161
unparsed entity information items, 162–163
Xmlns scheme (XPointer), 347–348
XML security specifications for, *see* Security, XML
XML Signatures, 383
XML Spy, 23–24
XML Writer, 23
XPath, 56–57. *See also* specific topics
definition of, 499
expressions in, 64–65
as language, 22
matching with, 22
nodes in, 63–72
context node, 65–71
current node, 71, 72
types, 63–64
syntax in, 57–60
Web sites, 490
in XSLT, 28
XPath 2.0, 479–486
accessors in, 485–486
data model for, 483–486
improvements to, 482–483
requirements for, 480–483
string content, manipulation of, 482
W3C Working Drafts of, 480
XML-schema typed content, manipulation of, 480–482
and XML standards, 482

XPath data model, 60–63, 137–164
and Document Object Model, 152–155
in-memory trees in, 138–152
result tree, 144–146
source tree, 138–143, 147
source/result documents, 145–146
and XML Information Set, 155–163
XPointer, 56, 333–351
data model for, 341–344
as fragment identifier language, 334
functions of, 348–350
and HTML fragment identifiers, 334–338
revisions of, 333
schemes of, 347–348
syntaxes of, 344–346
as technology beyond capabilities of XPath, 338–339
terminology used with, 339–341
uses of, 334
Web sites, 491
Xpointer scheme, 347
XQuery, 479, 486–487, 491–492
XSD Schema, 5
<xsl:apply-imports> element, 36
<xsl:apply-templates> element, 36–37
<xsl:attribute> element, 37–38
<xsl:attribute-set> element, 30
<xsl:call-template> element, 38–39
<xsl:choose> element, 39–40
<xsl:comment> element, 40
<xsl:copy> element, 40–41, 270–272
<xsl:copy-of> element, 41, 273–274
<xsl:decimal-format> element, 30–31
XSL dialect, 29
<xsl:element> element, 41–42, 267–270
<xsl:fallback> element, 42–43
XSL-FO, 356, 492
<xsl:for-each> element, 43
<xsl:if> element, 43–46, 292
<xsl:import> element, 30
<xsl:include> element, 31
<xsl:key> element, 31
<xsl:namespace-alias> element, 31–32
<xsl:number> element, 46
<xsl:otherwise> element, 47
<xsl:output> element, 32–33, 262–267
<xsl:param> element, 33–34
<xsl:preserve-space> element, 34
<xsl:processing-instruction> element, 47
<xsl:sort> element, 47
<xsl:strip-space> element, 34, 244
<xsl:stylesheet> element, 28–29
<xsl:template> element, 34–35, 284, 287
<xsl:text> element, 48
<xsl:value-of> element, 48, 284, 285
<xsl:variable> element, 35–36
<xsl:when> element, 48
<xsl:with-param> element, 48
XSLT, *see* Extensible Stylesheet Language Transformations
X-Smiles browser, 374–375

Z

Zip files, 51